Comparative Study of Programming Languages

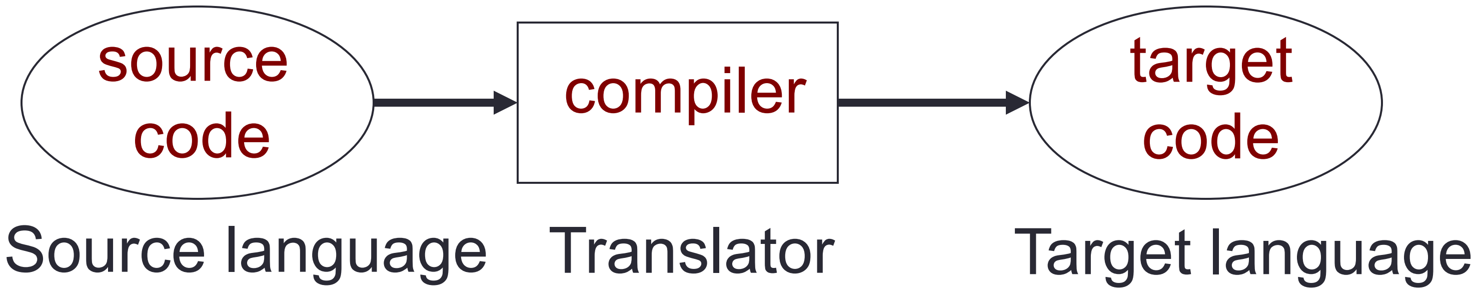
Program Evaluation

Learning Objectives

- Steps involved in program evaluation
- Compilation and interpretation
- Type systems
- Memory management

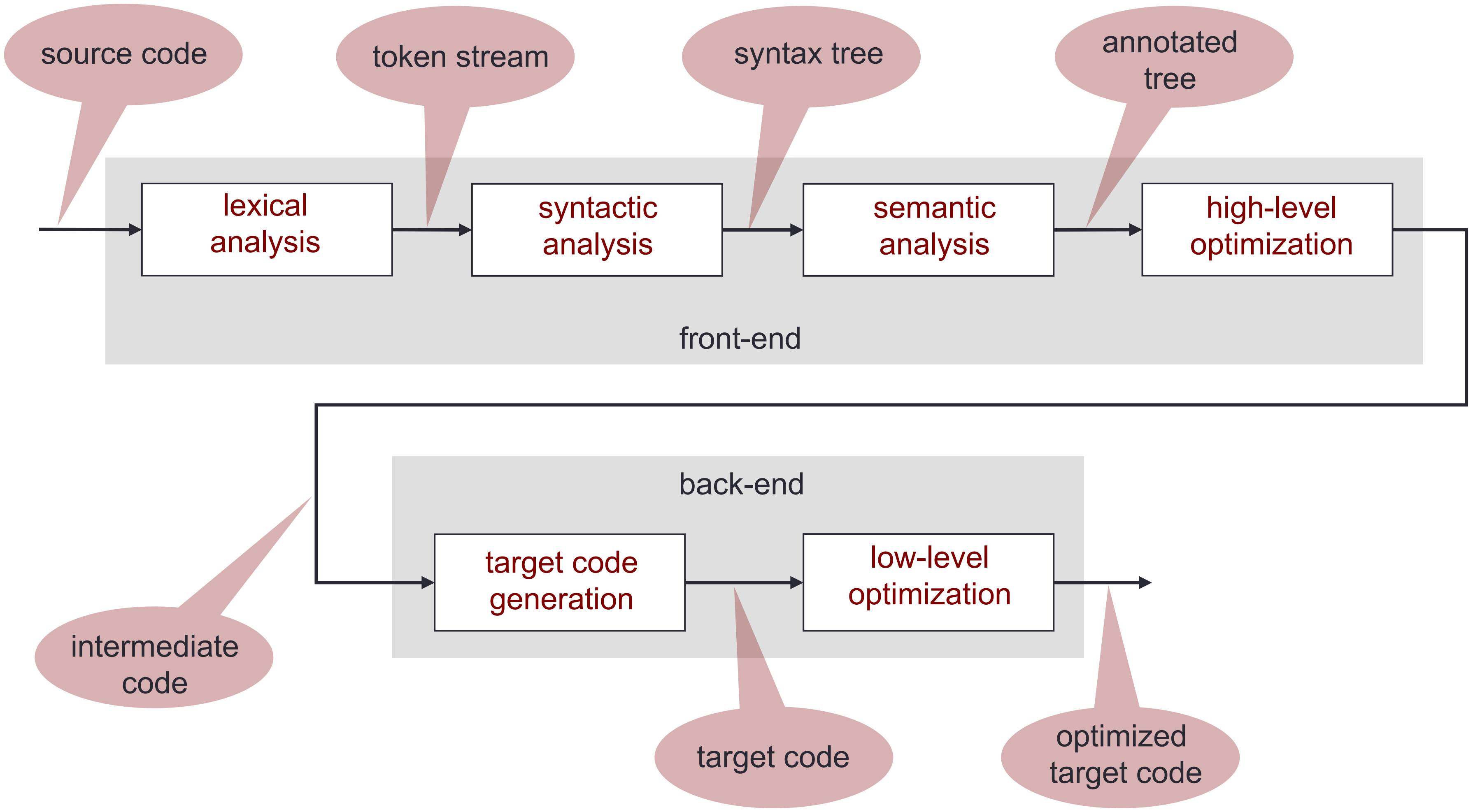
Introduction to Program Translation

- A compiler is a *translation system*.
- It translates programs written in a high level language into a lower level language, generally machine (binary) language.



PROGRAM TRANSLATION PROCESS

Phases of Program Translation



Front End

- The front end analyzes the source code to build an internal representation of the program, called the **intermediate representation**.
- It also manages the **symbol table**, a data structure mapping each symbol in the source code to associated information such as location, type and scope.

- The front-end is composed of: Lexical, Syntactic, Semantic analysis and High-level optimization.

- In most compilers, most of the front-end is driven by the Syntactic analyzer.
- It calls the Lexical analyzer for tokens and generates an *abstract syntax tree* when syntactic elements are recognized.
- The generated tree (or other intermediate representation) is then analyzed and optimized in a separate process.

- It has **no concern with the target machine**.

Back End

- The term back end is sometimes confused with code generator because of the overlapped functionality of generating assembly/machine code. Some literature uses middle end to distinguish the generic analysis and optimization phases in the back end from the machine-dependent code generators.

- The back-end is composed of: Code generation and low-level optimization.
- Uses the **intermediate representation** generated by the front-end to generate target machine code.

- **Heavily dependent on the target machine**.
- **Independent on the programming language compiled**.

Lexical Analysis

- Lexical analysis is the process of converting a sequence of characters into a sequence of **tokens**.

- A program or function which performs lexical analysis is called a lexical analyzer, lexer or scanner.
- A lexer often exists as a single function which is called by the parser.
- The lexical specification of a programming language is defined by a set of rules which defines the lexer, which are understood by a lexical analyzer generator such as lex or flex.
- The lexical analyzer (either generated automatically by a tool like lex, or hand-crafted) reads in a stream of characters, identifies the lexemes in the stream, categorizes them into tokens, and outputs a token stream.

- This is called "tokenizing." If the lexer finds an invalid token, it will report a lexical error.

Syntactic Analysis

- Syntax analysis involves **parsing** the token sequence to identify the syntactic structure of the program.

- The parser's output is some form of **intermediate representation** of the program's structure, typically a **parse tree**, which replaces the linear sequence of tokens with a tree structure built according to the rules of a formal grammar which define the language's syntax.
- This is usually done with reference to a **context-free grammar** which recursively defines components that can make up an expression and the order in which they must appear.

- The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.
- Parsers are written by hand or generated by parser generators, such as Yacc, Bison, or JavaCC.

Semantic Analysis

- Semantic analysis is the phase in which the compiler adds **semantic information** to the parse tree and builds the **symbol table**.
- This phase performs semantic checks such as **type checking** (checking for type errors), or **static object binding** (associating variable and function references with their definitions), or **definite assignment** (requiring all local variables to be initialized before use), rejecting incorrect programs or issuing warnings.
- Semantic analysis usually requires a complete parse tree, meaning that this phase logically follows the parsing phase, and logically precedes the code generation phase, though it is often possible to fold multiple phases into one pass over the code in a compiler implementation.
- Not all rules defining programming languages can be expressed by context-free grammars alone, for example semantic validity such as type validity and proper declaration of identifiers.
- These rules can be formally expressed with **attribute grammars** that implement **attribute migration** across syntax tree nodes when necessary.

High-Level Optimization

- High-level optimization involves an analysis phase for the gathering of program information from the intermediate representation derived by the front end.
- Typical analyzes are **data flow analysis** to build use-define chains, **dependence analysis**, **alias analysis**, **pointer analysis**, etc.
- Accurate analysis is the basis for any compiler optimization.
- The **call graph** and **control flow** graph are usually also built during the analysis phase.
- After analysis, the intermediate language representation is transformed into functionally equivalent but faster (or smaller) forms.
- Popular optimizations are **inline expansion**, **dead code elimination**, **constant propagation**, **loop transformation**, **register allocation** or even **automatic parallelization**.
- At this level, all activities are target machine language independent.

Code Generation

- The transformed intermediate language is **translated into the output language**, usually the native machine language of the system.

- This involves resource and storage decisions, such as deciding which variables to fit into registers and memory and the selection and scheduling of appropriate machine instructions along with their associated addressing modes.

PROGRAM TRANSLATION/EXECUTION MODELS

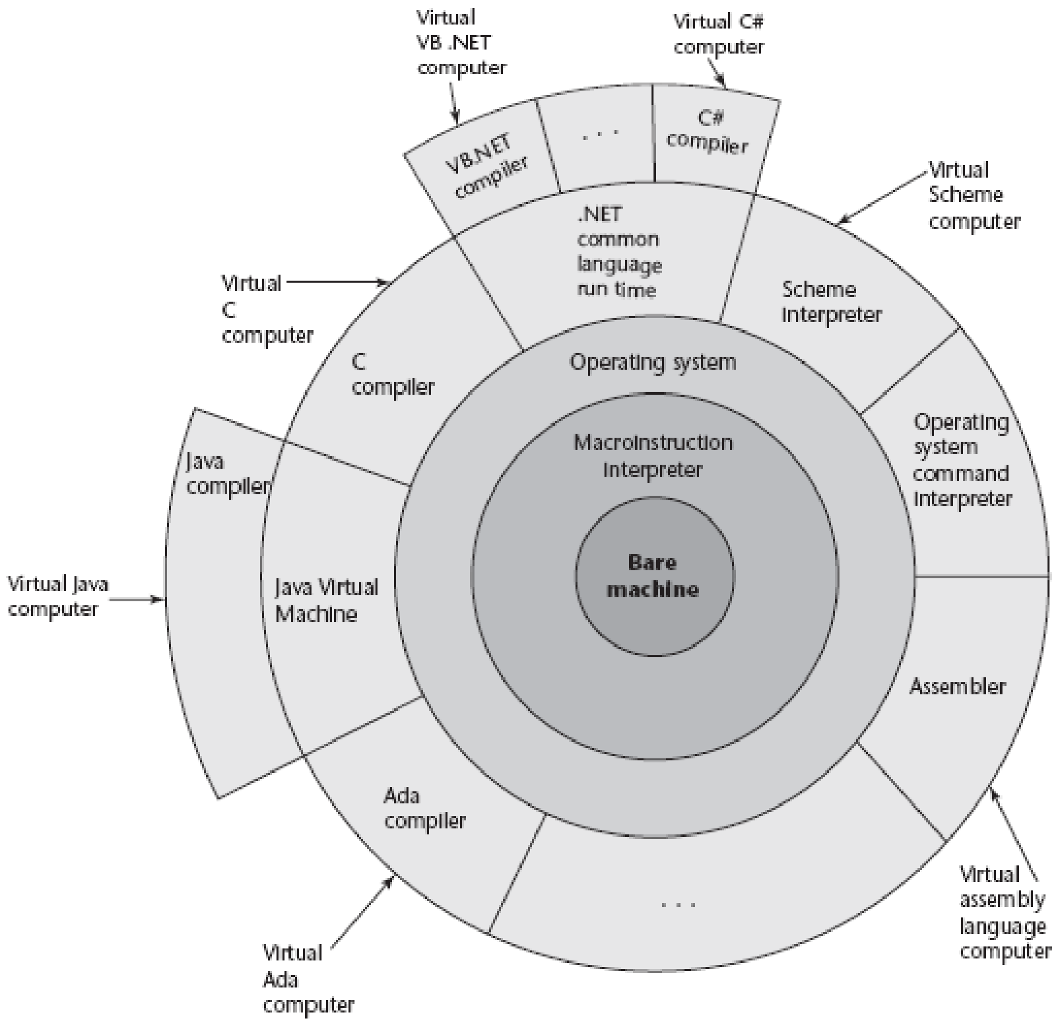
Three models

- Programming languages can be implemented (i.e. translated/executed) by any of the following three general methods:
  - **Compilation**: programs are translated directly into machine language by a compiler, the output of which is later executed directly by a computer.
  - **Interpretation**: programs are interpreted (i.e. simultaneously translated and executed) by an interpreter.
  - **Compilation/Interpretation Hybrid**: programs are first translated into an intermediate representation by a compiler, the output of which is later executed by and interpreter.
- No matter what model is used, all the language translation steps explained before are applied, except at different times.
- Languages that delay much of the checking to run-time are called “dynamic languages”.

Layered architecture view

Figure 1.2

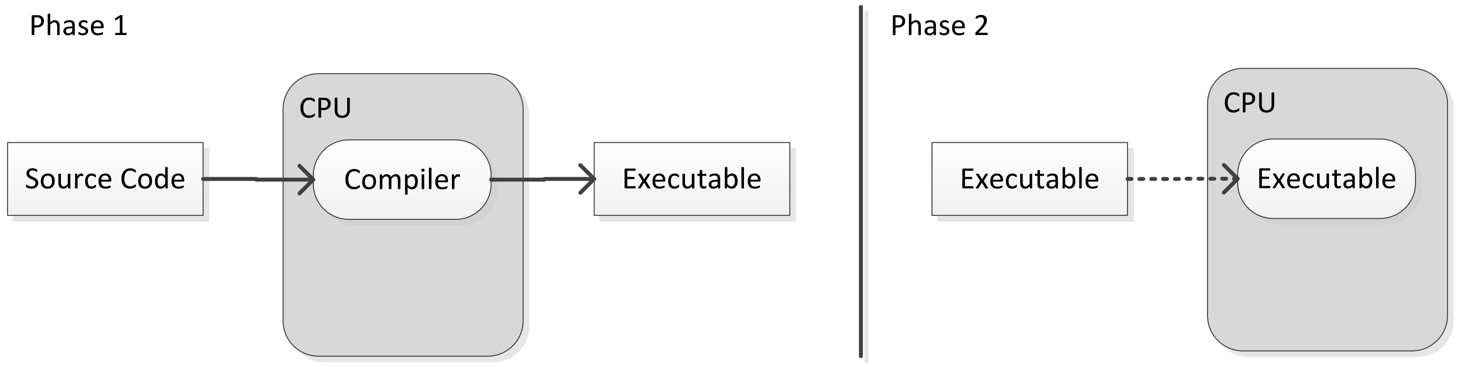
Layered interface of virtual computers, provided by a typical computer system



Compilation

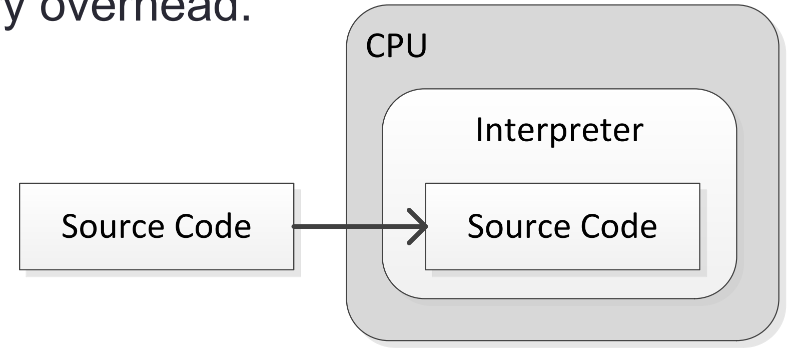
- A **compiler** is a computer program (or set of programs) that transforms source code written in a computer language (the source language) into another computer language (the target language, often having a binary form known as object code).

- The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a lower level language (e.g., assembly language or machine code) **before execution time**.



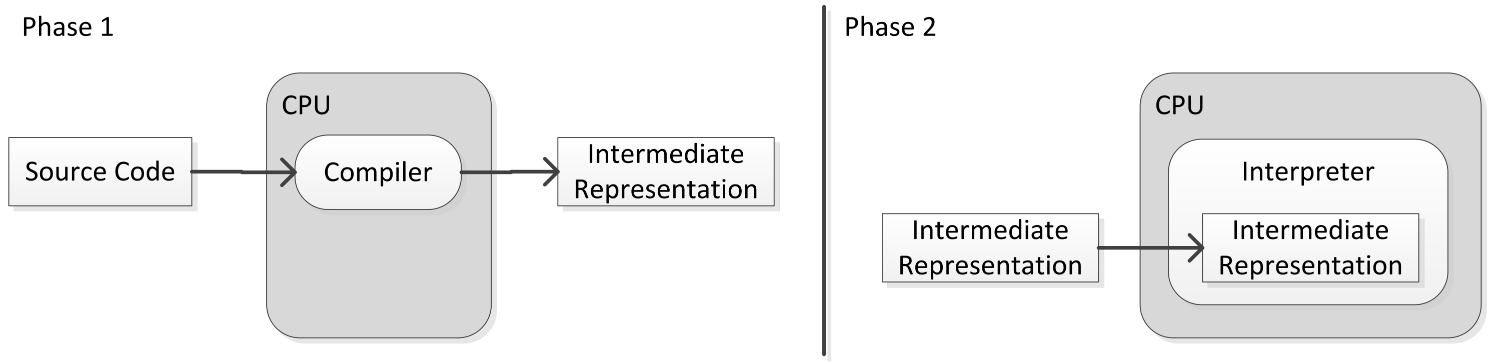
Interpretation

- An interpreted language is a programming language whose programs are not directly executed by the host CPU but rather executed (or said to be interpreted) by a software program known as an interpreter.
- Initially, interpreted languages were compiled line-by-line; that is, each line was compiled as it was about to be executed, and if a loop or subroutine caused certain lines to be executed multiple times, they would be recompiled every time. This has become much less common.
- In this case, all the program analysis phases must be happening at run-time, which can lead to unnecessary overhead.



Hybrid Compilation/Interpretation

- Nowadays, most interpreted languages use **an intermediate representation**, as a bridge between compilation and interpretation.
- In this case, a compiler may output some form of bytecode or threaded code, which is then executed by an interpreter. Examples include Python, Java, and Ruby.
- The intermediate representation can be compiled once and for all (as in Java), each time before execution (as in Perl or Ruby), or each time a change in the source is detected before execution (as in Python).


Phase 1
CPU
Source Code
Compiler
Intermediate Representation
Phase 2
CPU
Interpreter
Intermediate Representation
Intermediate Representation

Hybrid Compilation/Interpretation

- The source code of the program is often translated to a form that is more **convenient** to interpret, which may be some form of machine language for a virtual machine (such as Java's bytecode).
- An important and **time-consuming** part of the source code analysis is done **before** the interpretation starts.
- Some program **errors** are detected **before** execution starts.

Which One is better?

- In the early days of computing, language design was heavily influenced by the decision to use compilation or interpretation as a mode of execution.
- For example, some compiled languages require that programs must explicitly state the data type of a variable at the time it is declared or first used while some interpreted languages take advantage of the dynamic aspects of interpretation to make such declarations unnecessary.
- Such language issues delay the possibility of the binding mechanism to run-time. This does not mean, however, that all other program analysis must also be delayed to run-time.
- For example, Smalltalk 80, which was designed to be interpreted at run-time by an object-oriented interpreter, allows generic Objects created at compile-time to dynamically interact with each other at run-time.

Which One is better?

- Theoretically, any language may be compiled or interpreted, so this designation is applied purely because of common implementation practice and not some underlying property of a language.

- Many languages have been implemented using both compilers and interpreters, including Lisp, Pascal, C, BASIC, and Python.

Which One is better?

- Interpreting a language gives implementations some additional **flexibility** over compiled implementations. Features that are often easier to implement in interpreters than in compilers include (but are not limited to):
  - platform independence (Java's byte code, for example)
  - reflection and reflective usage of the evaluator
  - dynamic typing
  - dynamic scoping
- The main disadvantage of interpreting is a slower speed of program execution compared to direct machine code execution on the host CPU. A technique used to improve performance is just-in-time compilation which converts frequently executed sequences of interpreted instruction to host machine code.

Platforms

- The Microsoft .NET languages compile to CIL (Microsoft's Common Intermediate Language) which is often then compiled into native machine code; however there is a virtual machine capable of interpreting CIL.
- Many Lisp implementations can freely mix interpreted and compiled code. These implementations also use a compiler that can translate arbitrary source code at runtime to machine code.
- The Zend Engine for PHP.
- The Java Virtual Machine, which executes Java bytecode. Many other languages have been provided with compilers to generate bytecode so that they can use the Java Virtual Machine for execution. However, these are external projects, unlike for Microsoft's .NET and CIL, which integrates many languages.

Just-in-time Compilation

- Just-in-time compilation (JIT), also known as **dynamic translation**, is a method to improve the runtime performance of computer programs interpretation.
- Interpretation translates from a high-level language to a machine code continuously **during** every execution, whereas compilation translates into machine code **before** execution, and only requires this translation once.
- JIT compilers represent a **hybrid** approach, with translation occurring continuously, as with interpreters, but with **caching** of translated code to minimize performance degradation.
- It also offers other advantages over statically compiled code at development time, such as easier handling of late-bound data types and the ability to enforce some higher-level security guarantees.

Just-in-time Compilation

- Several modern runtime environments, such as Microsoft's .NET Framework and most implementations of Java, rely on JIT compilation for high-speed code execution.

- A just-in-time compiler can be used as a way to speed up execution of bytecode. At the time the bytecode is run, the just-in-time compiler will compile some or all of it to native machine code for better performance.

- This can be done per-file, per-function or even on any arbitrary code fragment.

Just-in-time Compilation

- A common goal of using JIT techniques is to reach or surpass the performance of static compilation, while maintaining some of the advantages of bytecode interpretation.
- Much of the "heavy lifting" of parsing the original source code and performing basic optimization is often handled at compile time, prior to deployment: compilation from bytecode to machine code is much faster than compiling from source.
- The deployed bytecode is portable, unlike native code.
- Compilers from bytecode to machine code are easier to write, because the portable bytecode compiler has already done much of the analysis work.

Just-in-time Compilation

- JIT techniques generally offers far better performance than interpreters. In addition, it can in some or many cases offer better performance than static compilation, as many optimizations are only feasible at run-time, such as:
  - The compilation can be optimized to the targeted CPU and the operating system model where the application runs.
  - The system is able to collect statistics about how the program is actually running in the environment it is in, and it can rearrange and recompile for optimum performance, i.e. adaptive optimization. One example of this is the Java HotSpot Performance Engine.
  - The system can do global code optimizations (e.g. inlining of library functions) without losing the advantages of dynamic linking and without the overheads inherent to static compilers and linkers.
  - As statically compiled garbage collected languages, a bytecode system can more easily rearrange memory for better cache utilization.

Ahead-of-Time Compilation

- Ahead-of-time compilation (AOT) refers to the act of compiling an intermediate language, such as Java bytecode or .NET Common Intermediate Language (CIL), into a system-dependent binary.
- Most languages that can be compiled to an intermediate language (such as bytecode) take advantage of just-in-time compilation.
- JIT compiles intermediate code into binary code for a native run while the intermediate code is executing, which may decrease an application's performance.
- Ahead-of-time compilation eliminates the need for this step by performing the compilation before execution rather than during execution.
- Using such a facility invariably means that some flexibility is lost, such as portability for example.

TYPE CHECKING

Type checking

- The process of verifying and enforcing the constraints of types – type checking – may occur either at compile-time (**static type checking**) or run-time (**dynamic type checking**). Most programming languages' implementations use a combination of the two approaches.

- If a language's type specifications are strict and enforces its typing rules strongly (e.g. allowing only type conversions which do not lose information or lead to run-time type or memory management errors), one can refer to the process as **strongly typed**, if not, as **weakly typed**.

- Strong typing leads to more reliable programs, as it allows more type errors to be caught before execution through static typing. It also leads to better run-time efficiency, as type binding can be done prior to execution.

Static type checking

- Static typing is a **limited** form of program verification. It allows many type errors to be caught early in the development cycle.

- Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for **all possible executions** of the program, which eliminates the need to repeat certain type checks every time the program is executed.

- Program execution may thus be made **more efficient** (i.e. faster or taking reduced memory) by omitting runtime type checks and enabling other optimizations.

- Statically typed languages include Ada, C, C++, C#, F#, Java, Fortran, Haskell, ML, Pascal, Perl, Objective-C and Scala.

Static type checking

- Because they evaluate type information during compilation, and therefore lack type information that is only available at run-time, static type checkers are **conservative**.
- They will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed. For example, even if an expression <complex test> always evaluates to true at run-time, a program containing the code

```
if <complex test> then 42 else <type error>
```

- will be rejected as ill-typed, because a static analysis cannot determine that the else branch won't be taken.
- The conservative behavior of static type checkers is advantageous when <complex test> evaluates to false infrequently: A static type checker can detect type errors in rarely used code paths.

Static type checking

- However, most widely used statically typed languages are not formally strongly typed.

- Most have "loopholes" in the programming language’s type specification enabling programmers to write code that circumvents the verification performed by a static type checker.

- For example, most C-style languages have **type coercion** that allows the programmer to enforce a change of type onto a value, and **union types** that allows the programmer to declare variables that are weakly typed.

Dynamic type checking

- A programming language is said to be dynamically typed when the majority of its type checking is performed at run-time as opposed to at compile-time.
- In strictly dynamic typing, **values have types but variables do not**. That is, a variable may refer to a value of any type.
- Dynamically typed language systems, compared to their statically typed cousins, make fewer compile-time checks on the source code (but will check, for example, that the program is syntactically correct).
- Compared to static typing, dynamic typing can be more flexible (e.g. by allowing programs to generate types and functionality based on run-time data), though **at the expense of reliability** due to lack of compile-time type checking.

Dynamic type checking

- Dynamic typing errors result **in runtime type errors**—that is, at runtime, a variable may hold a value of a certain type, and an operation nonsensical for that type is applied to it.
- This operation may occur long after the place where the programming mistake was made—that is, the place where the wrong type of data was passed into a variable. This may make the bug difficult to locate.
- These are clear advantages of statically typed languages have over languages that are strictly dynamically typed.

Dynamic type checking

- On the plus side, run-time checks can potentially be more sophisticated, since they can use dynamic information as well as any information that was present during compilation (such as in **dynamic binding** to allow **polymorphism**).

- On the other hand, runtime checks only assert that conditions hold in a particular execution of the program, and these checks are repeated for every execution of the program.

- Development in dynamically typed languages is often supported by programming practices such as **unit testing**. Testing is a key practice in professional software development, and is particularly important in dynamically typed languages.

- Dynamically typed languages include Erlang, Groovy, JavaScript, Lisp, Objective-C, Perl (with respect to user-defined types but not built-in types), PHP, Prolog, Python, Ruby and Smalltalk.

Duck typing

- Duck typing is a style of **object-oriented dynamic typing** in which an object's current set of methods and properties determines the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface.
- The name of the concept refers to the duck test, attributed to American writer James Whitcomb Riley, which may be phrased as follows:

"When I see a bird that walks like a duck and swims like a duck and quacks like a duck,
I call that bird a duck."

Duck typing

- In duck typing, one is concerned with the aspects of an object that are **used**, rather than with the type definition of the object itself.

- For example, in a non-duck-typed language, one can create an object of type **`Duck`** and call that object's **`walk()`** and **`quack()`** methods, which are explicitly declared in type **`Duck`**'s declaration. Calling **`walk()`** or **`quack()`** on an object of a type that does not explicitly declare such methods would result in a compile-time error.

- In a duck-typed language, one could create an object of any type and call that object's **`walk()`** and **`quack()`** methods. If the object does not have the methods that are called then the function signals a run-time error.

- Duck typing is not restrained by the type of arguments in method declarations or calls or, by extension, to type verifications in function bodies. It rather relies on documentation, clear code, and testing to ensure correct use.

Duck typing

- Consider the following pseudo-code for a duck typed language:

```
function calculate(a, b, c) => return (a+b)*c
example1 = calculate (1, 2, 3)
example2 = calculate ([1, 2, 3], [4, 5, 6], 2)
example3 = calculate ('apples ', 'and oranges, ', 3)
print to_string example1
print to_string example2
print to_string example3
```

Duck typing

- Each time the **`calculate()`** function is called, objects without related inheritance may be used e.g. numbers, lists and strings. As long as the objects passed as parameters support the "+" and "*" methods, the operation will succeed. Execution of the preceding code may result in:

```
9
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
apples and oranges, apples and oranges, apples and
oranges
```

- Thus, duck typing **allows polymorphism without inheritance**. The only requirement that function **`calculate`** needs in its variables is having the "+" and the "*" methods defined.
- Few languages actually use duck typing, including Ruby and Python, as well as C# though the use of the **`dynamic`** parameter type declaration.

Structural typing

- A structural type system (or **property-based** type system) is a major class of type system, in which type compatibility and equivalence are determined by the types' **structure** or **composition**, and **not through explicit declarations**.
- Structural type systems are used to determine if types are **equivalent**, as well as if a type is a **subtype** of another.
- It contrasts with **nominative systems**, where comparisons are based on explicit declarations or the names of the types, and **duck typing**, in which only the part of the structure accessed at runtime is checked for compatibility.

Structural typing

- In structural typing, two values are considered to have compatible types if their types have identical structure, including the structure of its “membership”.

- Depending on the semantics of the language, this generally means that for each **feature** within a type, there must be a corresponding and identical feature in the other type.

- Some languages may differ on the details, such as whether the features must match in name, or only in their individual type specifications.

Structural typing

- ML and Objective Caml are examples of structurally-typed languages.
- C++ template functions exhibit structural typing on type arguments.
-
- Structural typing may also apply to languages that support **subtype polymorphism**. In this case, one type is a subtype of another if and only if it contains all the features of the base type; the subtype may contain additional features.
- The slicing problem in C++ is a good example of a problem related to structural typing.
- A pitfall of structural typing versus nominative typing is that two separately defined types intended for different purposes, each consisting of a pair of numbers, could be considered the same type by the type system, simply because they happen to have identical structure.

Nominal typing

- A nominal or nominative type system (or name-based type system) is a major class of type system, in which compatibility and equivalence of data types is determined by **explicit declarations** and/or the **names** of the types.
- Nominative systems are used to determine if types are equivalent, as well as if a type is a subtype of another.
- Nominal typing means that two variables are type-compatible if and only if their declarations name the same type. For example, in C, two **`struct`** types with different names are never considered compatible, even if they have identical field declarations.
- However, C also allows a **`typedef`** declaration, which introduces an alias for an existing type. These are merely syntactical and do not differentiate the type from its alias for the purpose of type checking.

Nominal typing

- In a similar fashion, nominal subtyping means that one type is a subtype of another if and only if it is explicitly declared to be so in its definition.
- Nominally-typed languages typically enforce the requirement that declared subtypes be structurally compatible. However, subtypes which are structurally compatible "by accident", but not declared as subtypes, are not considered to be subtypes.
- C, C++, C# and Java all primarily use both nominal typing and nominal subtyping.
- Nominal typing is useful at preventing accidental type equivalence, and is considered to have better type-safety than structural typing. The cost is a reduced flexibility, as, for example, nominal typing does not allow new super-types to be created without modification of the existing subtypes.

MEMORY MANAGEMENT

Memory management

- Memory management is the act of managing the computer memory allocated to a program as it executes.

- This involves providing ways to **allocate** portions of memory to programs when they need it, and **freeing** it for reuse when no longer needed.

- As memory is a finite resource, the management and optimization of memory usage is critical to a computer system.

Memory management

- Some of the memory allocation can be **statically** done at compile-time for variables that are known to exist for the entire running of the program, and whose size, inferred from their types, can be determined at compile time.
- However, as in the case of pointer variables, often the size of memory to be allocated **cannot be determined** at compile time.
- By extension, and particularly if recursion is allowed, memory allocation for function calls cannot be determined at compile time.
- These require more elaborated runtime memory allocation schemes in place in the **runtime system** executing the programs.

Memory management

- In most modern programming languages, memory is allocated either on the **heap** or the **stack**. Memory allocated on the stack stores **local variables**, parameters, and return values during function calls.

- These are variables that have a **lifetime** predetermined by their **scope**. Stack-based memory allocation is simple and is generally managed by the **run-time system** in close collaboration with the **operating system**.

- Memory allocated on the heap is handled differently by different programming languages. In C and C++, memory must be managed explicitly, while in C# and Java, memory is implicitly allocated and deallocated through an automated process part of the run-time system called **garbage collection**.

Memory management

- No matter what scheme is used, memory allocation and deallocation is ultimately handled through interaction with the the **operating system**.

- Additional memory management mechanisms such as virtual memory management, memory paging and memory swapping are implemented at the operating system level and are transparent to the programming language implementation.

- The runtime system associated with the execution of programs makes simple calls to allocate/deallocate memory to the operating system independently of memory management strategies used by the operating system.

Memory management

- Even though manual (i.e. explicit) memory management is easier for the language runtime system, it adds **complexity** for the programmer, which can lead to **bugs** and coding errors due to improper memory management and object lifecycle maintenance.
- Even though garbage collection is **more demanding** for the language runtime system, it reduces the complexity for the programmer, and helps reduce the number of bugs and coding errors caused by manual memory management.

Garbage collection

- Garbage collection is a form of **automatic memory management**. It is a special case of **resource management**, in which the limited resource being managed is memory.
- The garbage collector is named as such as it attempts to reclaim “garbage”, or memory occupied by objects that are no longer in use by the program.
- However, the term “garbage collection” globally refers also to the allocation of memory that was previously reclaimed.
- Garbage collection was invented by **John McCarthy** around 1959 to allocate/deallocate memory for list structures is **LISP**, whose size is inherently variant.

Garbage collection

- Garbage collection is often portrayed as the **opposite** of manual memory management, which requires the programmer to specify which objects to deallocate and return to the memory system.
- However, many systems use a **combination of various approaches**, and other techniques such as stack allocation and region inference using syntactical program blocks are also used to manage memory.

Garbage collection

- Garbage collection does not traditionally manage limited **resources** other than memory that typical programs use, such as network sockets, database handles, user interaction windows, and file and device descriptors.

- Methods used to manage such resources, particularly **constructors** and **destructors**, may suffice as well to manage memory, leaving no need for garbage collection.
-
- Some garbage collection systems allow such other resources to be associated with a region of memory that, when collected, causes the other resource to be reclaimed. This is called **finalization**.

Garbage collection

- The basic principles of garbage collection are:
  - Find data objects in a program that cannot be accessed in the future
  - Reclaim the resources used by those objects
- By making manual memory deallocation unnecessary (and often forbidding it), garbage collection frees the programmer from having to worry about releasing objects that are no longer needed, which can otherwise consume a significant amount of **design effort**.
- It also aids programmers in their efforts to make programs more **stable**, because it prevents many runtime errors.
- For example, it prevents **dangling pointer** errors, where a reference to a deallocated object is used.

Garbage collection: benefits

- Frees the programmer from manually dealing with memory allocation and deallocation. As a result, certain categories of **bugs are eliminated** or substantially reduced:
  - **Dangling pointer** bugs, which occur when a piece of memory is freed while there are still pointers to it, and one of those pointers is then used.
  - **Double free** bugs, which occur when the program attempts to free a region of memory that has already been freed.
  - Certain kinds of **memory leaks**, in which a program fails to free memory occupied by objects that will not be used again, leading, over time, to memory exhaustion.
- Not having to deal with memory allocation also saves substantial development time, and yields more **programming productivity**.

Garbage collection: drawbacks

- Garbage collection is a process that consumes computing resources in deciding what memory is to be freed and when, by reconstructing facts that may have been known to the programmer.
- The penalty for the convenience of not annotating memory usage manually in the code is overhead leading, potentially, to **decreased performance**.
- The point when the garbage is actually collected can be **unpredictable**, resulting in delays scattered throughout a session.
- Unpredictable delays can be **unacceptable in real-time environments** such as device drivers, or transaction processing.

Garbage collection: drawbacks

- Garbage collectors generally can do nothing about **logical memory leaks**, where a region of memory is still referenced but is never actually used.

- Perhaps the most significant problem is that programs that rely on garbage collectors often interact badly with cache and virtual memory systems, as they typically reserve more address space than the program actually uses at any one time, and may touch otherwise idle pages. This can lead the computer/operating system to start “thrashing”, i.e. spending more time copying data between various storage areas than performing useful work.

- This may make it impossible for a programmer to reason about the performance effects of design choices, making **performance tuning difficult**.

- This also leads garbage-collecting programs to interfere with **other programs** competing for resources.

Garbage collection: drawbacks

- The execution of a program using a garbage collector is **not deterministic**. An object which becomes eligible for garbage collection will usually be cleaned up eventually, but there is no guarantee **when** (or even **if**) that will happen.
- Most run-time systems using garbage collectors require manual deallocation of limited non-memory resources (using **finalizers**), as an automatic deallocation during the garbage collection phase may run too late or in the wrong circumstances.
- Also, the performance impact caused by the garbage collector is seemingly random and hard to predict, leading to program execution performance that is non-deterministic.

Garbage collection: finalizers

- A finalizer is a special method that is executed when an object is garbage collected. It is similar in function to a **destructor**.
- A finalizer is a piece of code that ensures that certain necessary actions are taken when an acquired resource (such as a file or access to a hardware device) is no longer being used. This could be closing the file or signalling to the operating system that the hardware device is no longer needed.
- Unlike destructors, finalizers are not deterministic. A destructor is run when the program explicitly frees an object. A finalizer, by contrast, is executed when the internal garbage collection system frees the object.
- Programming languages which use finalizers include Java and C#. In C#, and a few others which support finalizers, the syntax for declaring a finalizer mimics that of destructors in C++.

Garbage collection: finalizers

- Due to the lack of programmer control over their execution, it is usually recommended to avoid finalizers for any but the most trivial operations.

- In particular, operations often performed in destructors are not usually appropriate for finalizers. For example, destructors are often used to free expensive resources such as file or network handles. If placed in a finalizer, the resources may remain in use for long periods of time after the program is finished with them.

- Instead, most languages encourage the **dispose pattern** whereby the object has a method to clean up the object's resources, leaving the finalizer as a fail-safe in the case where the dispose method doesn't get called. The C# language supports the dispose pattern explicitly, via the **`IDisposable`** interface.

Garbage collection

- Many programming languages **require** garbage collection, either as part of the language specification (e.g., Java, C#, and most scripting languages) or effectively for practical implementation (e.g., formal languages like lambda calculus); these are said to be **garbage collected languages**.
- Other languages were designed for use with manual memory management, but have garbage collected implementations **available** (e.g., C, C++).
- Some languages, like Ada, Modula-3, and C++ allow both garbage collection and manual memory management to **co-exis**t in the same application by using separate heaps for collected and manually managed objects; others, like D, are garbage collected but allow the user to manually delete objects and also entirely disable garbage collection when speed is required.

Garbage collection

- Generally speaking, higher-level programming languages are more likely to have garbage collection as a standard feature. In languages that do not have built-in garbage collection, it can often be added through a library, as with the **Boehm garbage collector** for C and C++. This approach is not without drawbacks, such as changing object creation and destruction mechanisms.
- Most functional programming languages, such as ML, Haskell, and APL, have garbage collection built-in. Lisp, which introduced functional programming, is especially notable for introducing this mechanism.
- Other dynamic languages, such as Ruby (but not Perl, or PHP, which use reference counting), also tend to use garbage collection.
- Object-oriented programming languages such as Smalltalk and Java usually provide integrated garbage collection. A notable exception is C++ which, uniquely, relies on constructors and destructors.

Manual memory management

- Manual memory management refers to the usage of explicit programming instructions by the programmer to identify and deallocate unused objects.

- Up to the mid 1990s, the majority of languages used in industry supported manual memory management. Today, languages with garbage collection are becoming more popular; the main manually-managed languages still in widespread use today are C and C++.

- Such programming languages use **explicit code** to determine when to allocate new objects. C uses the **`malloc`** function; C++ and Java use the **`new`** operator; determining when an object ought to be created is unproblematic.

- The fundamental issue is the determination of when an object is no longer needed. In manual memory allocation, this is also specified manually by the programmer; via functions such as **`free`** in C, or the **`delete`** operator in C++.

Manual memory management

- Manual memory management is known to enable several major classes of bugs into a program, when used incorrectly:
  - When an unused object is never released back to the free store, this is known as a **memory leak**. In some cases, memory leaks may be tolerable, such as a program which "leaks" a bounded amount of memory over its lifetime, or a short-running program which relies on an operating system to deallocate its resources when it terminates. However, in many cases memory leaks occur in long-running programs, and in such cases an **unbounded** amount of memory is leaked. When this occurs, the size of the available free store continues to decrease over time; when it finally is exhausted the program then crashes.
  - When an object is **deleted more than once**, or when the programmer attempts to release a pointer to an object not allocated from the free store, catastrophic failure of the dynamic memory management system can result. The result of such actions can include heap corruption, premature destruction of a different (and newly-created) object which happens to occupy the same location in memory as the multiply-deleted object, and other forms of undefined behavior.
  - Pointers to deleted objects become **wild pointers** if used post-deletion; attempting to use such pointers can result in difficult-to-diagnose bugs.

Manual memory management

- Languages which exclusively use garbage collection are known to avoid the last two classes of defects.

- Memory leaks can still occur, but are generally less severe than memory leaks in manual memory allocation schemes.

- Manual memory management has one correctness advantage, which comes into play when objects own scarce system resources (like graphics resources, file handles, or database connections) which must be relinquished when an object is destroyed.

- Languages with manual management, via the use of **destructors**, can arrange for such actions to occur at the precise time of object destruction. In C++, this ability is put to further use to automate memory deallocation within an otherwise-manual framework. The use of the `auto_ptr` template in C++ STL to perform memory management is a common paradigm.

Manual memory management

- Many advocates of manual memory management argue that it affords **superior performance** when compared to automatic techniques such as garbage collection.
- Manual allocation does not suffer from the long and unpredictable "pause" times often associated with garbage collection.
- This is especially an issue in real time systems, where unbounded and unpredictable collection cycles are generally unacceptable.
- Manual allocation is also known to be more appropriate for systems where memory is a scarce resource.

Manual memory management

- On the other hand, manual management has documented performance disadvantages:
  - Calls to **`delete`** incur an overhead each time they are made, this overhead can be amortized in garbage collection cycles. This is especially true of multithreaded applications, where **`delete`** calls must be synchronized.
  - The allocation routine may be more complicated, and slower. Some garbage collection schemes, such as those with heap compaction, can maintain the free store as a simple array of memory (as opposed to the complicated implementations required by manual management schemes).