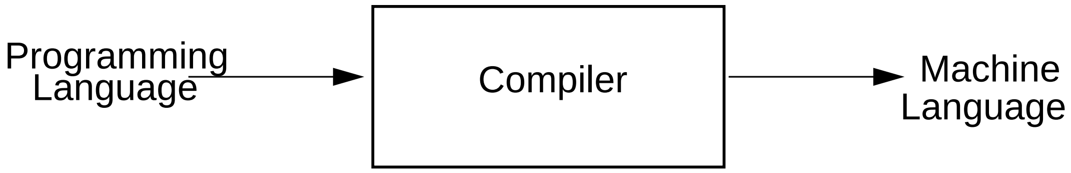
Compilers

Compilers are fundamental to modern computing.

They act as *translators*, transforming human-oriented *programming languages* into computer-oriented *machine languages.*

To most users, a compiler can be viewed as a “black box” that performs the transformation shown below.



A compiler allows programmers to ignore the machine-dependent details of programming.

Compilers allow programs and programming skills to be *machine-independent* and *platform-independent*.

Compilers also aid in detecting and correcting programming errors (which are all too common).

Compiler techniques also help to improve computer security. For example, the Java Bytecode Verifier helps to guarantee that Java security rules are satisfied.

Compilers currently help in protection of intellectual property (using *obfuscation*) and provenance (through *watermarking*).

History of Compilers

The term *compiler* was coined in the early 1950s by Grace Murray Hopper. Translation was viewed as the “compilation” of a sequence of machine-language subprograms selected from a library.

One of the first real compilers was the FORTRAN compiler of the late 1950s. It allowed a programmer to use a problem-oriented source language.

Ambitious “optimizations” were used to produce efficient machine code, which was vital for early computers with quite limited capabilities.

Efficient use of machine resources is still an essential requirement for modern compilers.

Compilers Enable Programming Languages

Programming languages are used for much more than “ordinary” computation.

- TeX and LaTeX use compilers to translate text and formatting commands into intricate typesetting commands.
- Postscript, generated by text-formatters like LaTeX, Word, and FrameMaker, is really a programming language. It is translated and executed by laser printers and document previewers to produce a readable form of a document. A standardized document representation language allows documents to be freely interchanged, independent of how

they were created and how they will be viewed.

- Mathmatica is an interactive system that intermixes programming with mathematics; it is possible to solve intricate problems in both symbolic and numeric form. This system relies heavily on compiler techniques to handle the specification, internal representation, and solution of problems.
- Verilog and VHDL support the creation of VLSI circuits. A *silicon compiler* specifies the layout and composition of a VLSI circuit mask, using standard cell designs. Just as an ordinary compiler understands and enforces programming language rules, a silicon compiler understands and enforces the design rules that dictate the feasibility of a given circuit.

- Interactive tools often need a programming language to support automatic analysis and modification of an artifact.
  How do you *automatically* filter or change a MS Word document? You need a text-based specification that can be processed, like a program, to check validity or produce an updated version.

When do We Run a Compiler?

- **Prior to execution**
  This is standard. We compile a program once, then use it repeatedly.
- **At the start of each execution**
  We can incorporate values known at the start of the run to improve performance.
  A program may be partially complied, then completed with values set at execution-time.
- **During execution**
  Unused code need not be compiled.
  Active or “hot” portions of a program may be specially optimized.
- **After execution**
  We can profile a program, looking for heavily used routines, which can be specially optimized for later runs.

What do Compilers Produce?

Pure Machine Code

Compilers may generate code for a particular machine, not assuming any operating system or library routines. This is “pure code” because it includes nothing beyond the instruction set. This form is rare; it is sometimes used with system implementation languages, that define operating systems or embedded applications (like a programmable controller). Pure code can execute on bare hardware without dependence on any other software.

Augmented Machine Code

Commonly, compilers generate code for a machine architecture *augmented* with operating system routines and run-time language support routines.

To use such a program, a particular operating system must be used and a collection of run-time support routines (I/O, storage allocation, mathematical functions, etc.) must be available. The combination of machine instruction and OS and run-time routines define a *virtual machine*—a computer that exists only as a hardware/ software combination.

Virtual Machine Code

Generated code can consist *entirely* of virtual instructions (no native code at all). This allows code to run on a variety of computers.

Java, with its JVM (Java Virtual Machine) is a great example of this approach.

If the virtual machine is kept simple and clean, its interpreter can be easy to write. Machine interpretation slows execution by a factor of 3:1 to perhaps 10:1 over compiled code.

A “Just in Time” (*JIT*) compiler can translate “hot” portions of virtual code into native code to speed execution.

Advantages of Virtual Instructions

Virtual instructions serve a variety of purposes.

- They simplify a compiler by providing suitable primitives (such as method calls, string manipulation, and so on).
- They aid compiler transportability.
- They may decrease in the size of generated code since instructions are designed to match a particular programming language (for example, JVM code for Java).

Almost all compilers, to a greater or lesser extent, generate code for a virtual machine, some of whose operations must be interpreted.

Formats of Translated Programs

Compilers differ in the format of the target code they generate. Target formats may be categorized as *assembly language*, *relocatable binary*, or *memory-image*.

- **Assembly Language (Symbolic) Format**

  A text file containing assembler source code is produced. A number of code generation decisions (jump targets, long vs. short address forms, and so on) can be left for the assembler. This approach is good for instructional projects.

Generating assembler code supports *cross-compilation* (running a compiler on one computer, while its target is a second computer). Generating assembly language also simplifies debugging and understanding a compiler (since you can see the generated code).

C (rather than a specific assembly language) can generated, treating C as a "universal assembly language."

C is far more machine-independent than any particular assembly language. However, some aspects of a program (such as the run-time representations of program and data) are inaccessible using C code, but readily accessible in assembly language.

- **Relocatable Binary Format**

  Target code may be generated in a *binary format* with external references and local instruction and data addresses are not yet bound. Instead, addresses are assigned relative to the beginning of the module or relative to symbolically named locations. A *linkage* step adds support libraries and other separately compiled routines and produces an absolute binary program format that is executable.

- **Memory-Image (Absolute Binary) Form**

  Compiled code may be loaded into memory and immediately executed. This is faster than going through the intermediate step of link/editing. The ability to access library and precompiled routines may be limited. The program must be recompiled for each execution. Memory-image compilers are useful for student and debugging use, where frequent changes are the rule and compilation costs far exceed execution costs.

Java is designed to use and share classes designed and implemented at a variety of sites. Rather than use a fixed copy of a class (which may be outdated), the JVM supports *dynamic linking* of externally defined classes. When first referenced, a class definition may be remotely fetched, checked, and loaded during program execution. In this way “foreign code” can be guaranteed to be up-to-date and secure.

The Structure of a Compiler

A compiler performs two major tasks:

- Analysis of the source program being compiled
- Synthesis of a target program

Almost all modern compilers are *syntax-directed:* The compilation process is driven by the syntactic structure of the source program.

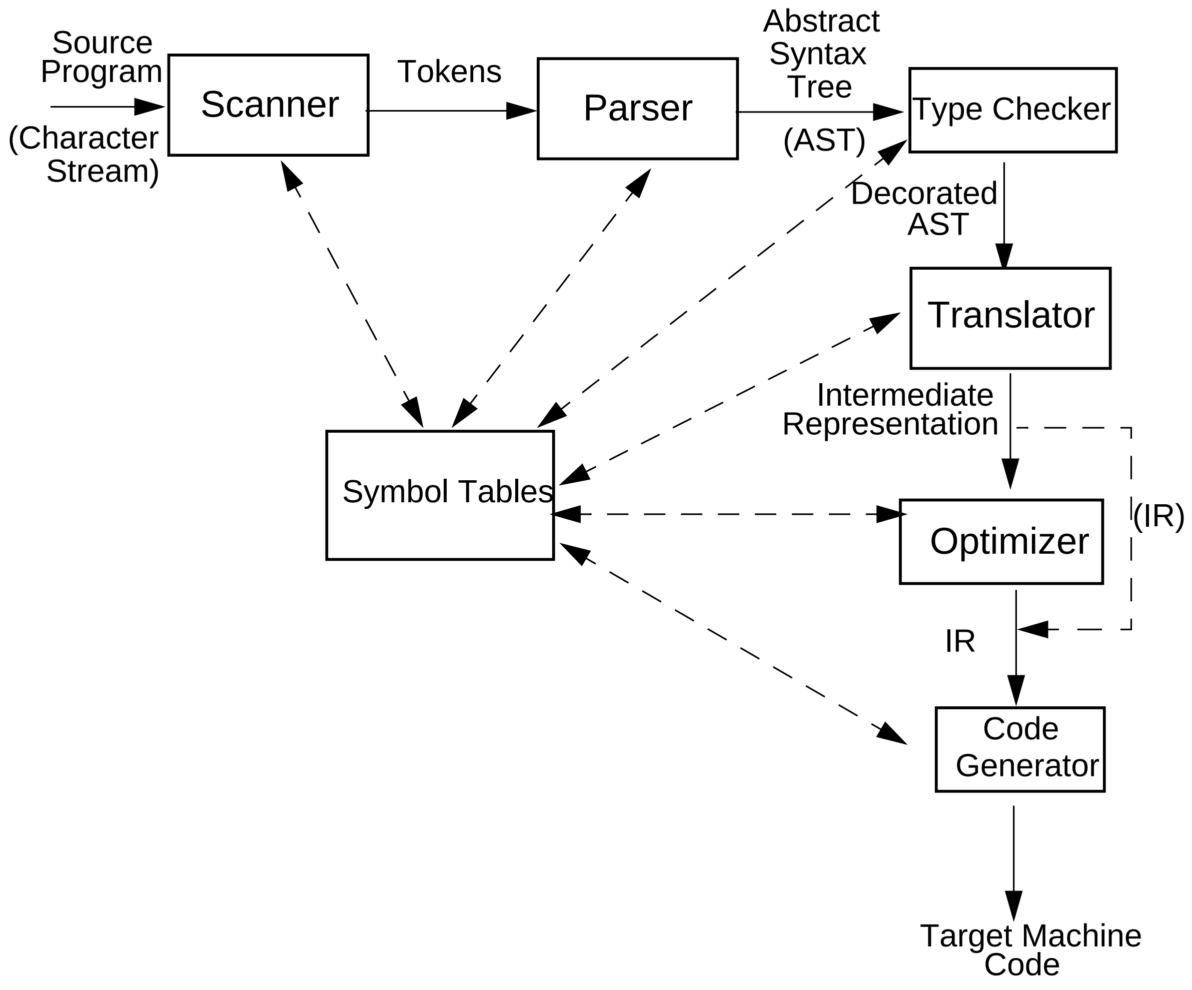
A parser builds semantic structure out of tokens, the elementary symbols of programming language syntax. Recognition of syntactic structure is a major part of the analysis task.

Semantic analysis examines the meaning (semantics) of the program. Semantic analysis plays a dual role.

It finishes the analysis task by performing a variety of correctness checks (for example, enforcing type and scope rules). Semantic analysis also begins the synthesis phase.

The synthesis phase may translate source programs into some intermediate representation (IR) or it may directly generate target code.

If an IR is generated, it then serves as input to a *code generator* component that produces the desired machine-language program. The IR may optionally be transformed by an *optimizer* so that a more efficient program may be generated.



The Structure of a Syntax-Directed Compiler