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 supports transportable code, that can 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, the interpreter can be quite easy to write. Machine interpretation slows execution speed 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 procedure calls, string manipulation, and so on).
- They contribute to 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, using 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 defined and implemented at a variety of organizations. 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.

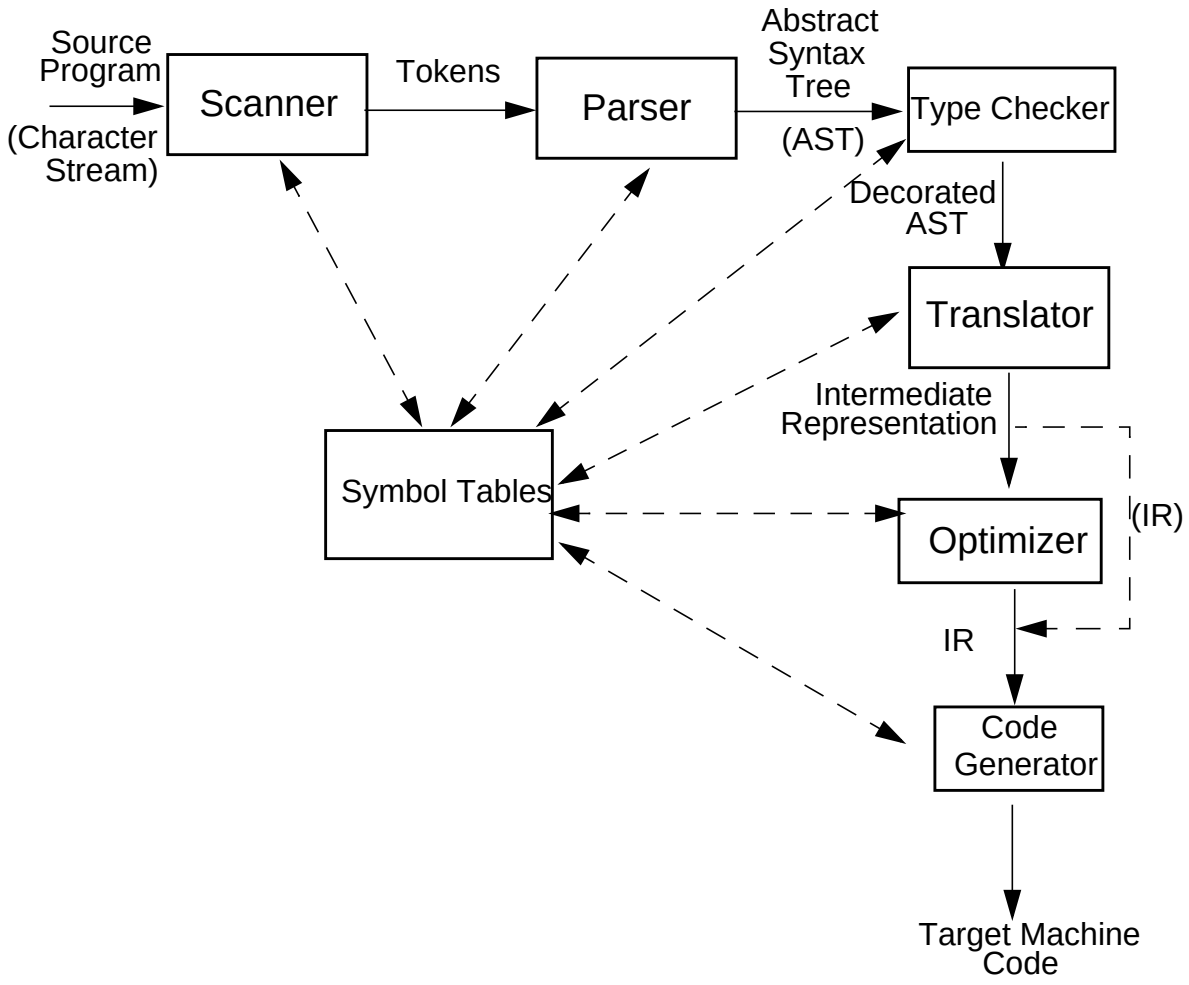
A parser builds semantic structure out of tokens, the elementary symbols of programming language syntax. Recognition of syntactic structure is a major part of the analysis task.

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

Scanner

The scanner reads the source program, character by character. It groups individual characters into tokens (identifiers, integers, reserved words, delimiters, and so on). When necessary, the actual character string comprising the token is also passed along for use by the semantic phases.

The scanner does the following:

- It puts the program into a compact and uniform format (a stream of tokens).
- It eliminates unneeded information (such as comments).
- It sometimes enters preliminary information into symbol tables (for

example, to register the presence of a particular label or identifier).

- It optionally formats and lists the source program

Building tokens is driven by token descriptions defined using *regular expression* notation.

Regular expressions are a formal notation able to describe the tokens used in modern programming languages. Moreover, they can drive the *automatic generation* of working scanners given only a specification of the tokens. Scanner generators (like Lex, Flex and Jlex) are valuable compiler-building tools.

Parser

Given a syntax specification (as a context-free grammar, CFG), the parser reads tokens and groups them into language structures.

Parsers are typically created from a CFG using a parser generator (like Yacc, Bison or Java CUP).

The parser verifies correct syntax and may issue a syntax error message.

As syntactic structure is recognized, the parser usually builds an abstract syntax tree (AST), a concise representation of program structure, which guides semantic processing.

Type Checker (Semantic Analysis)

The type checker checks the *static semantics* of each AST node. It verifies that the construct is legal and meaningful (that all identifiers involved are declared, that types are correct, and so on).

If the construct is semantically correct, the type checker "decorates" the AST node, adding type or symbol table information to it. If a semantic error is discovered, a suitable error message is issued.

Type checking is purely dependent on the semantic rules of the source language. It is independent of the compiler's target machine.

Translator (Program Synthesis)

If an AST node is semantically correct, it can be translated. Translation involves capturing the run-time "meaning" of a construct.

For example, an AST for a while loop contains two subtrees, one for the loop's control expression, and the other for the loop's body. *Nothing* in the AST shows that a while loop loops! This "meaning" is captured when a while loop's AST is translated. In the IR, the notion of testing the value of the loop control expression, and conditionally executing the loop body becomes explicit.

The translator is dictated by the semantics of the source language.

Little of the nature of the target machine need be made evident. Detailed information on the nature of the target machine (operations available, addressing, register characteristics, etc.) is reserved for the code generation phase.

In simple non-optimizing compilers (like our class project), the translator generates target code directly, without using an IR.

More elaborate compilers may first generate a high-level IR (that is source language oriented) and then subsequently translate it into a low-level IR (that is target machine oriented). This approach allows a cleaner separation of source and target dependencies.

Optimizer

The IR code generated by the translator is analyzed and transformed into functionally equivalent but improved IR code by the optimizer.

The term optimization is misleading: we don't always produce the best possible translation of a program, even after optimization by the best of compilers.

Why?

Some optimizations are *impossible* to do in all circumstances because they involve an undecidable problem. Eliminating unreachable ("dead") code is, in general, impossible.

Other optimizations are too expensive to do in all cases. These involve NP-complete problems, believed to be inherently exponential. Assigning registers to variables is an example of an NP-complete problem.

Optimization can be complex; it may involve numerous subphases, which may need to be applied more than once.

Optimizations may be turned off to speed translation. Nonetheless, a well designed optimizer can significantly speed program execution by simplifying, moving or eliminating unneeded computations.

Code Generator

IR code produced by the translator is mapped into target machine code by the code generator. This phase uses detailed information about the target machine and includes machine-specific optimizations like *register allocation* and *code scheduling*.

Code generators can be quite complex since good target code requires consideration of many special cases.

Automatic generation of code generators is possible. The basic approach is to match a low-level IR to target instruction templates, choosing instructions which best match each IR instruction.

A well-known compiler using automatic code generation

techniques is the GNU C compiler. GCC is a heavily optimizing compiler with machine description files for over ten popular computer architectures, and at least two language front ends (C and C++).

Symbol Tables

A symbol table allows information to be associated with identifiers and shared among compiler phases. Each time an identifier is used, a symbol table provides access to the information collected about the identifier when its declaration was processed.