WIKIPEDIA

Compiler

A **compiler** is a computer program that translates computer code written in one programming language (the source language) into another programming language (the target language). 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, object code, or machine code) to create an executable program.[1][2]:p1

However, there are many different types of compilers. If the compiled program can run on a computer whose CPU or operating system is different from the one on which the compiler runs, the compiler is a cross-compiler. A bootstrap compiler is written in the language that it intends to compile. A program that translates from a low-level language to a higher level one is a decompiler. A program that translates between high-level languages is usually called a source-to-source compiler or transpiler. A language rewriter is usually a program that translates the form of expressions without a change of language. The term compiler-compiler refers to tools used to create parsers that perform syntax analysis.

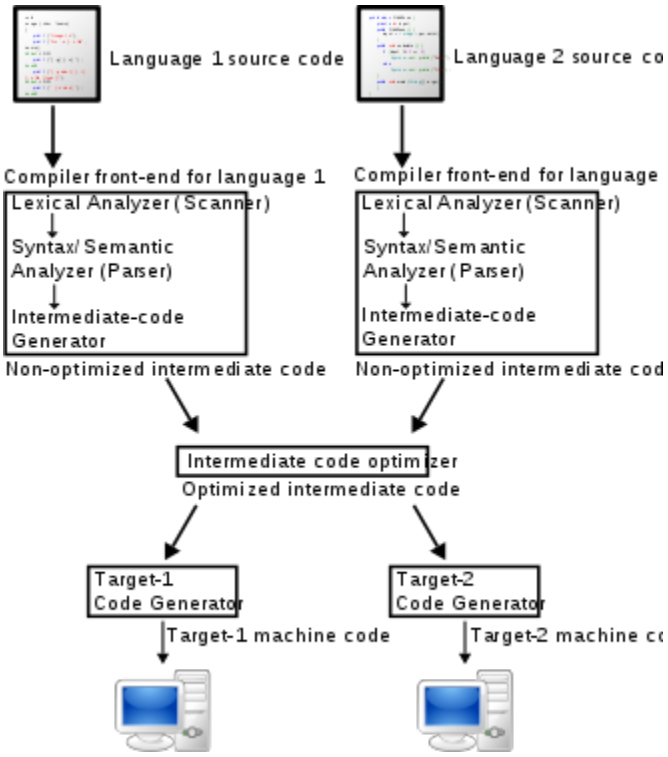
A compiler is likely to perform many or all of the following operations: preprocessing, lexical analysis, parsing, semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and code generation. Compilers implement these operations in phases that promote efficient design and correct transformations of source input to target output. Program faults caused by incorrect compiler behavior can be very difficult to track down and work around; therefore, compiler implementers invest significant effort to ensure compiler correctness.[3]

Compilers are not the only language processor used to transform source programs. An interpreter is computer software that transforms and then executes the indicated operations.[2]:p2 The translation process influences the design of computer languages which leads to a preference of compilation or interpretation. In practice, an interpreter can be implemented for compiled languages and compilers can be implemented for interpreted languages.

Contents

History

Theoretical computing concepts developed by scientists, mathematicians, and engineers formed the basis of digital modern computing development during World War II. Primitive binary languages evolved because digital devices only understand ones and zeros and the circuit patterns in the underlying machine architecture. In the late 1940s, assembly languages were created to offer a more workable abstraction of the computer architectures. Limited memory capacity of early computers led to substantial technical challenges when the first compilers were designed. Therefore, the compilation process needed to be divided into several small programs. The front end programs produce the analysis products used by the back end programs to generate target code. As computer technology provided more resources, compiler designs could align better with the compilation process.



A diagram of the operation of a typical multi-language, multi-target compiler

It is usually more productive for a programmer to use a high-level language, so the development of high-level languages followed naturally from the capabilities offered by digital computers. High-level languages are formal languages that are strictly defined by their syntax and semantics which form the high-level language architecture. Elements of these formal languages include:

- *Alphabet*, any finite set of symbols;
- *String*, a finite sequence of symbols;
- *Language*, any set of strings on an alphabet.

The sentences in a language may be defined by a set of rules called a grammar.[4]

Backus–Naur form (BNF) describes the syntax of "sentences" of a language and was used for the syntax of Algol 60 by John Backus.[5] The ideas derive from the context-free grammar concepts by Noam Chomsky, a linguist.[6] "BNF and its

extensions have become standard tools for describing the syntax of programming notations, and in many cases parts of compilers are generated automatically from a BNF description."[7]

In the 1940s, Konrad Zuse designed an algorithmic programming language called Plankalkül ("Plan Calculus"). While no actual implementation occurred until the 1970s, it presented concepts later seen in APL designed by Ken Iverson in the late 1950s.[8] APL is a language for mathematical computations.

High-level language design during the formative years of digital computing provided useful programming tools for a variety of applications:

- FORTRAN (Formula Translation) for engineering and science applications is considered to be the first high-level language.[9]
- COBOL (Common Business-Oriented Language) evolved from A-0 and FLOW-MATIC to become the dominant high-level language for business applications.[10]
- LISP (List Processor) for symbolic computation.[11]

Compiler technology evolved from the need for a strictly defined transformation of the high-level source program into a low-level target program for the digital computer. The compiler could be viewed as a front end to deal with the analysis of the source code and a back end to synthesize the analysis into the target code. Optimization between the front end and back end could produce more efficient target code.[12]

Some early milestones in the development of compiler technology:

- **1952** – An Autocode compiler developed by Alick Glennie for the Manchester Mark I computer at the University of Manchester is considered by some to be the first compiled programming language.
- **1952** – Grace Hopper's team at Remington Rand wrote the compiler for the A-0 programming language (and coined the term *compiler* to describe it),[13][14] although the A-0 compiler functioned more as a loader or linker than the modern notion of a full compiler.
- **1954-1957** – A team led by John Backus at IBM developed FORTRAN which is usually considered the first high-level language. In 1957, they completed a FORTRAN compiler that is generally credited as having introduced the first unambiguously complete compiler.
- **1959** – The Conference on Data Systems Language (CODASYL) initiated development of COBOL. The COBOL design drew on A-0 and FLOW-MATIC. By the early 1960s COBOL was compiled on multiple architectures.
- **1958-1962** – John McCarthy at MIT designed LISP.[15] The symbol processing capabilities provided useful features for artificial intelligence research. In 1962, LISP 1.5 release noted some tools: an interpreter written by Stephen Russell and Daniel J. Edwards, a compiler and assembler written by Tim Hart and Mike Levin.[16]

Early operating systems and software were written in assembly language. In the 60s and early 70s, the use of high-level languages for system programming was still controversial due to resource limitations. However, several research and industry efforts began the shift toward high-level systems programming languages, for example, BCPL, BLISS, B, and C.

BCPL (Basic Combined Programming Language) designed in 1966 by Martin Richards at the University of Cambridge was originally developed as a compiler writing tool.[17] Several compilers have been implemented, Richards' book provides insights to the language and its compiler.[18] BCPL was not only an influential systems programming language

that is still used in research[19] but also provided a basis for the design of B and C languages.

BLISS (Basic Language for Implementation of System Software) was developed for a Digital Equipment Corporation (DEC) PDP-10 computer by W.A. Wulf's Carnegie Mellon University (CMU) research team. The CMU team went on to develop BLISS-11 compiler one year later in 1970.

Multics (Multiplexed Information and Computing Service), a time-sharing operating system project, involved MIT, Bell Labs, General Electric (later Honeywell) and was led by Fernando Corbató from MIT.[20] Multics was written in the PL/I language developed by IBM and IBM User Group.[21] IBM's goal was to satisfy business, scientific, and systems programming requirements. There were other languages that could have been considered but PL/I offered the most complete solution even though it had not been implemented.[22] For the first few years of the Mulitics project, a subset of the language could be compiled to assembly language with the Early PL/I (EPL) compiler by Doug McIlory and Bob Morris from Bell Labs.[23] EPL supported the project until a boot-strapping compiler for the full PL/I could be developed.[24]

Bell Labs left the Multics project in 1969: "Over time, hope was replaced by frustration as the group effort initially failed to produce an economically useful system."[25] Continued participation would drive up project support costs. So researchers turned to other development efforts. A system programming language B based on BCPL concepts was written by Dennis Ritchie and Ken Thompson. Ritchie created a boot-strapping compiler for B and wrote Unics (Uniplexed Information and Computing Service) operating system for a PDP-7 in B. Unics eventually became spelled Unix.

Bell Labs started development and expansion of C based on B and BCPL. The BCPL compiler had been transported to Multics by Bell Labs and BCPL was a preferred language at Bell Labs.[26] Initially, a front-end program to Bell Labs' B compiler was used while a C compiler was developed. In 1971, a new PDP-11 provided the resource to define extensions to B and rewrite the compiler. By 1973 the design of C language was essentially complete and the Unix kernel for a PDP-11 was rewritten in C. Steve Johnson started development of Portable C Compiler (PCC) to support retargeting of C compilers to new machines.[27][28]

Object-oriented programming (OOP) offered some interesting possibilities for application development and maintenance. OOP concepts go further back but were part of LISP and Simula language science.[29] At Bell Labs, the development of C++ became interested in OOP.[30] C++ was first used in 1980 for systems programming. The initial design leveraged C language systems programming capabilities with Simula concepts. Object-oriented facilities were added in 1983.[31] The Cfront program implemented a C++ front-end for C84 language compiler. In subsequent years several C++ compilers were developed as C++ popularity grew.

In many application domains, the idea of using a higher-level language quickly caught on. Because of the expanding functionality supported by newer programming languages and the increasing complexity of computer architectures, compilers became more complex.

DARPA (Defense Advanced Research Projects Agency) sponsored a compiler project with Wulf's CMU research team in 1970. The Production Quality Compiler-Compiler PQCC design would produce a Production Quality Compiler (PQC) from formal definitions of source language and the target.[32] PQCC tried to extend the term compiler-compiler beyond the traditional meaning as a parser generator (e.g., Yacc) without much success. PQCC might more properly be referred to

as a compiler generator.

PQCC research into code generation process sought to build a truly automatic compiler-writing system. The effort discovered and designed the phase structure of the PQC. The BLISS-11 compiler provided the initial structure.[33] The phases included analyses (front end), intermediate translation to virtual machine (middle end), and translation to the target (back end). TCOL was developed for the PQCC research to handle language specific constructs in the intermediate representation.[34] Variations of TCOL supported various languages. The PQCC project investigated techniques of automated compiler construction. The design concepts proved useful in optimizing compilers and compilers for the object-oriented programming language Ada.

The Ada Stoneman Document formalized the program support environment (APSE) along with the kernel (KAPSE) and minimal (MAPSE). An Ada interpreter NYU/ED supported development and standardization efforts with the American National Standards Institute (ANSI) and the International Standards Organization (ISO). Initial Ada compiler development by the U.S. Military Services included the compilers in a complete integrated design environment along the lines of the Stoneman Document. Army and Navy worked on the Ada Language System (ALS) project targeted to DEC/VAX architecture while the Air Force started on the Ada Integrated Environment (AIE) targeted to IBM 370 series. While the projects did not provide the desired results, they did contribute to the overal effort on Ada development.[35]

Other Ada compiler efforts got underway in Britain at the University of York and in Germany at the University of Karlsruhe. In the U. S., Verdix (later acquired by Rational) delivered the Verdix Ada Development System (VADS) to the Army. VADS provided a set of development tools including a compiler. Unix/VADS could be hosted on a variety of Unix platforms such as DEC Ultrix and the Sun 3/60 Solaris targeted to Motorola 68020 in an Army CECOM evaluation.[36] There were soon many Ada compilers available that passed the Ada Validation tests. The Free Software Foundation GNU project developed the GNU Compiler Collection (GCC) which provides a core capability to support multiple languages and targets. The Ada version GNAT is one of the most widely used Ada compilers. GNAT is free but there is also commercial support, for example, AdaCore, was founded in 1994 to provide commercial software solutions for Ada. GNAT Pro includes the GNU GCC based GNAT with a tool suite to provide an integrated development environment.

High-level languages continued to drive compiler research and development. Focus areas included optimization and automatic code generation. Trends in programming languages and development environments influenced compiler technology. More compilers became included in language distributions (PERL, Java Development Kit) and as a component of an IDE (VADS, Eclipse, Ada Pro). The interrelationship and interdependence of technologies grew. The advent of web services promoted growth of web languages and scripting languages. Scripts trace back to the early days of Command Line Interfaces (CLI) where the user could enter commands to be executed by the system. User Shell concepts developed with languages to write shell programs. Early Windows designs offered a simple batch programming capability. The conventional transformation of these language used an interpreter. While not widely used, Bash and Batch compilers have been written. More recently sophisticated interpreted languages became part of the developers tool kit. Modern scripting languages include PHP, Python, Ruby and Lua. (Lua is widely used in game development.) All of these have interpreter and compiler support.[37]

"When the field of compiling began in the late 50s, its focus was limited to the translation of high-level language programs into machine code ... The compiler field is increasingly intertwined with other disciplines including computer architecture, programming languages, formal methods, software engineering, and computer security."[38] The "Compiler Research: The

Next 50 Years" article noted the importance of object-oriented languages and Java. Security and parallel computing were cited among the future research targets.

Compiler construction

A compiler implements a formal transformation from a high-level source program to a low-level target program. Compiler design can define an end to end solution or tackle a defined subset that interfaces with other compilation tools e.g. preprocessors, assemblers, linkers. Design requirements include rigorously defined interfaces both internally between compiler components and externally between supporting toolsets.

In the early days, the approach taken to compiler design was directly affected by the complexity of the computer language to be processed, the experience of the person(s) designing it, and the resources available. Resource limitations led to the need to pass through the source code more than once.

A compiler for a relatively simple language written by one person might be a single, monolithic piece of software. However, as the source language grows in complexity the design may be split into a number of interdependent phases. Separate phases provide design improvements that focus development on the functions in the compilation process.

One-pass versus multi-pass compilers

Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing lots of work and early computers did not have enough memory to contain one program that did all of this work. So compilers were split up into smaller programs which each made a pass over the source (or some representation of it) performing some of the required analysis and translations.

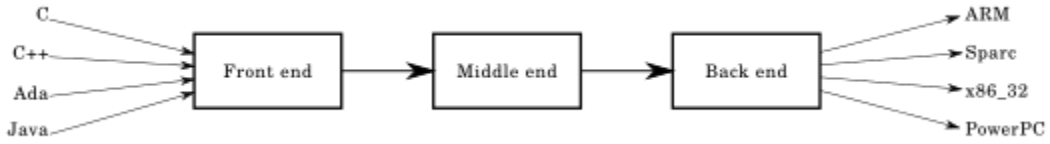
The ability to compile in a single pass has classically been seen as a benefit because it simplifies the job of writing a compiler and one-pass compilers generally perform compilations faster than multi-pass compilers. Thus, partly driven by the resource limitations of early systems, many early languages were specifically designed so that they could be compiled in a single pass (e.g., Pascal).

In some cases the design of a language feature may require a compiler to perform more than one pass over the source. For instance, consider a declaration appearing on line 20 of the source which affects the translation of a statement appearing on line 10. In this case, the first pass needs to gather information about declarations appearing after statements that they affect, with the actual translation happening during a subsequent pass.

The disadvantage of compiling in a single pass is that it is not possible to perform many of the sophisticated optimizations needed to generate high quality code. It can be difficult to count exactly how many passes an optimizing compiler makes. For instance, different phases of optimization may analyse one expression many times but only analyse another expression once.

Splitting a compiler up into small programs is a technique used by researchers interested in producing provably correct compilers. Proving the correctness of a set of small programs often requires less effort than proving the correctness of a larger, single, equivalent program.

Three-stage compiler structure



Compiler design

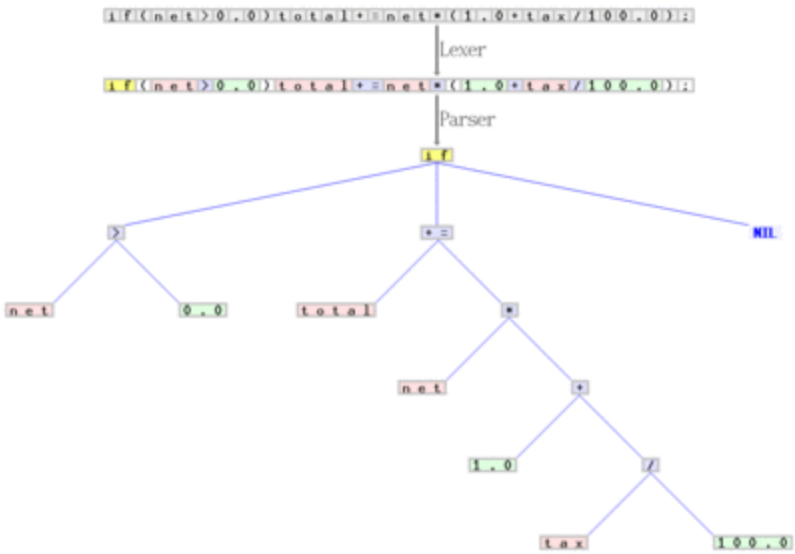
Regardless of the exact number of phases in the compiler design, the phases can be assigned to one of three stages. The stages include a front end, a middle end, and a back end.

- The *front end* verifies syntax and semantics according to a specific source language. For statically typed languages it performs type checking by collecting type information. If the input program is syntactically incorrect or has a type error, it generates errors and warnings, highlighting them on the source code. Aspects of the front end include lexical analysis, syntax analysis, and semantic analysis. The front end transforms the input program into an intermediate representation (IR) for further processing by the middle end. This IR is usually a lower-level representation of the program with respect to the source code.
- The *middle end* performs optimizations on the IR that are independent of the CPU architecture being targeted. This source code/machine code independence is intended to enable generic optimizations to be shared between versions of the compiler supporting different languages and target processors. Examples of middle end optimizations are removal of useless (dead code elimination) or unreachable code (reachability analysis), discovery and propagation of constant values (constant propagation), relocation of computation to a less frequently executed place (e.g., out of a loop), or specialization of computation based on the context. Eventually producing the "optimized" IR that is used by the back end.
- The *back end* takes the optimized IR from the middle end. It may perform more analysis, transformations and optimizations that are specific for the target CPU architecture. The back end generates the target-dependent assembly code, performing register allocation in the process. The back end performs instruction scheduling, which re-orders instructions to keep parallel execution units busy by filling delay slots. Although most algorithms for optimization are NP-hard, heuristic techniques are well-developed and currently implemented in production-quality compilers. Typically the output of a back end is machine code specialized for a particular processor and operating system.

This front/middle/back-end approach makes it possible to combine front ends for different languages with back ends for different CPUs while sharing the optimizations of the middle end.[39] Practical examples of this approach are the GNU Compiler Collection, LLVM,[40] and the Amsterdam Compiler Kit, which have multiple front-ends, shared optimizations and multiple back-ends.

Front end

The front end analyzes the source code to build an internal representation of the program, called the intermediate representation (IR). 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.



Lexer and parser example for C. Starting from the sequence of characters `"if(net>0.0)total+=net*(1.0+tax/100.0);"`, the scanner composes a sequence of tokens, and categorizes each of them, for example as identifier, reserved word, number literal, or operator. The latter sequence is transformed by the parser into a syntax tree, which is then treated by the remaining compiler phases. The scanner and parser handles the regular and properly context-free parts of the grammar for C, respectively.

While the frontend can be a single monolithic function or program, as in a scannerless parser, it is more commonly implemented and analyzed as several phases, which may execute sequentially or concurrently. This method is favored due to its modularity and separation of concerns. Most commonly today, the frontend is broken into three phases: lexical analysis (also known as lexing), syntax analysis (also known as scanning or parsing), and semantic analysis. Lexing and parsing comprise the syntactic analysis (word syntax and phrase syntax, respectively), and in simple cases these modules (the lexer and parser) can be automatically generated from a grammar for the language, though in more complex cases these require manual modification. The lexical grammar and phrase grammar are usually context-free grammars, which simplifies analysis significantly, with context-sensitivity handled at the semantic analysis phase. The semantic analysis phase is generally more complex and written by hand, but can be partially or fully automated using attribute grammars. These phases themselves can be further broken down: lexing as scanning and evaluating, and parsing as building a concrete syntax tree (CST, parse tree) and then transforming it into an abstract syntax tree (AST, syntax tree). In some cases additional phases are used, notably *line reconstruction* and *preprocessing,* but these are rare.

The main phases of the front end include the following:

- *Line reconstruction* converts the input character sequence to a canonical form ready for the parser. Languages which strop their keywords or allow arbitrary spaces within identifiers require this phase. The top-down, recursive-descent, table-driven parsers used in the 1960s typically read the source one character at a time and did not require a separate tokenizing phase. Atlas Autocode and Imp (and some implementations of ALGOL and Coral 66) are examples of stropped languages whose compilers would have a *Line Reconstruction* phase.
- *Preprocessing* supports macro substitution and conditional compilation. Typically the preprocessing phase occurs before syntactic or semantic analysis; e.g. in the case of C, the preprocessor manipulates lexical tokens rather than syntactic forms. However, some languages such as Scheme support macro substitutions based on syntactic forms.
- *Lexical analysis* (also known as *lexing* or *tokenization*) breaks the source code text into a

sequence of small pieces called *lexical tokens*.[41] This phase can be divided into two stages: the *scanning*, which segments the input text into syntactic units called *lexemes* and assign them a category; and the *evaluating*, which converts lexemes into a processed value. A token is a pair consisting of a *token name* and an optional *token value*.[42] Common token categories may include identifiers, keywords, separators, operators, literals and comments, although the set of token categories varies in different programming languages. The lexeme syntax is typically a regular language, so a finite state automaton constructed from a regular expression can be used to recognize it. The software doing lexical analysis is called a lexical analyzer. This may not be a separate step—it can be combined with the parsing step in scannerless parsing, in which case parsing is done at the character level, not the token level.

- *Syntax analysis* (also known as *parsing*) involves parsing the token sequence to identify the syntactic structure of the program. This phase typically builds 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. The parse tree is often analyzed, augmented, and transformed by later phases in the compiler.[43]
- *Semantic analysis* 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 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.

Middle end

The middle end, also known as *optimizer*, performs optimizations on the intermediate representation in order to improve the performance and the quality of the produced machine code.[44] The middle end contains those optimizations that are independent of the CPU architecture being targeted.

The main phases of the middle end include the following:

- Analysis: This is the gathering of program information from the intermediate representation derived from the input; data-flow analysis is used to build use-define chains, together with dependence analysis, alias analysis, pointer analysis, escape analysis, etc. Accurate analysis is the basis for any compiler optimization. The control flow graph of every compiled function and the call graph of the program are usually also built during the analysis phase.
- Optimization: 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 and even automatic parallelization.

Compiler analysis is the prerequisite for any compiler optimization, and they tightly work together. For example, dependence analysis is crucial for loop transformation.

The scope of compiler analysis and optimizations vary greatly; their scope may range from operating within a basic block, to whole procedures, or even the whole program. There is a trade-off between the granularity of the optimizations and the cost of compilation. For example, peephole optimizations are fast to perform during compilation but only affect a small

local fragment of the code, and can be performed independently of the context in which the code fragment appears. In contrast, interprocedural optimization requires more compilation time and memory space, but enable optimizations which are only possible by considering the behavior of multiple functions simultaneously.

Interprocedural analysis and optimizations are common in modern commercial compilers from HP, IBM, SGI, Intel, Microsoft, and Sun Microsystems. The free software GCC was criticized for a long time for lacking powerful interprocedural optimizations, but it is changing in this respect. Another open source compiler with full analysis and optimization infrastructure is Open64, which is used by many organizations for research and commercial purposes.

Due to the extra time and space needed for compiler analysis and optimizations, some compilers skip them by default. Users have to use compilation options to explicitly tell the compiler which optimizations should be enabled.

Back end

The back end is responsible for the CPU architecture specific optimizations and for code generation[44].

The main phases of the back end include the following:

- *Machine dependent optimizations*: optimizations that depend on the details of the CPU architecture that the compiler targets.[45] A prominent example is peephole optimizations, which rewrites short sequences of assembler instructions into more efficient instructions.
- *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 (see also Sethi-Ullman algorithm). Debug data may also need to be generated to facilitate debugging.

Compiler correctness

Compiler correctness is the branch of software engineering that deals with trying to show that a compiler behaves according to its language specification.[46] Techniques include developing the compiler using formal methods and using rigorous testing (often called compiler validation) on an existing compiler.

Compiled versus interpreted languages

Higher-level programming languages usually appear with a type of translation in mind: either designed as compiled language or interpreted language. However, in practice there is rarely anything about a language that *requires* it to be exclusively compiled or exclusively interpreted, although it is possible to design languages that rely on re-interpretation at run time. The categorization usually reflects the most popular or widespread implementations of a language — for instance, BASIC is sometimes called an interpreted language, and C a compiled one, despite the existence of BASIC compilers and C interpreters.

Interpretation does not replace compilation completely. It only hides it from the user and makes it gradual. Even though an interpreter can itself be interpreted, a directly executed program is needed somewhere at the bottom of the stack (see

machine language).

Further, compilers can contain interpreters for optimization reasons. For example, where an expression can be executed during compilation and the results inserted into the output program, then it prevents it having to be recalculated each time the program runs, which can greatly speed up the final program. Modern trends toward just-in-time compilation and bytecode interpretation at times blur the traditional categorizations of compilers and interpreters even further.

Some language specifications spell out that implementations *must* include a compilation facility; for example, Common Lisp. However, there is nothing inherent in the definition of Common Lisp that stops it from being interpreted. Other languages have features that are very easy to implement in an interpreter, but make writing a compiler much harder; for example, APL, SNOBOL4, and many scripting languages allow programs to construct arbitrary source code at runtime with regular string operations, and then execute that code by passing it to a special evaluation function. To implement these features in a compiled language, programs must usually be shipped with a runtime library that includes a version of the compiler itself.

Types

One classification of compilers is by the platform on which their generated code executes. This is known as the *target platform.*

A *native* or *hosted* compiler is one whose output is intended to directly run on the same type of computer and operating system that the compiler itself runs on. The output of a cross compiler is designed to run on a different platform. Cross compilers are often used when developing software for embedded systems that are not intended to support a software development environment.

The output of a compiler that produces code for a virtual machine (VM) may or may not be executed on the same platform as the compiler that produced it. For this reason such compilers are not usually classified as native or cross compilers.

The lower level language that is the target of a compiler may itself be a high-level programming language. C, viewed by some as a sort of portable assembly language, is frequently the target language of such compilers. For example, Cfront, the original compiler for C++, used C as its target language. The C code generated by such a compiler is usually not intended to be readable and maintained by humans, so indent style and creating pretty C intermediate code are ignored. Some of the features of C that make it a good target language include the `#line` directive, which can be generated by the compiler to support debugging of the original source, and the wide platform support available with C compilers.

While a common compiler type outputs machine code, there are many other types:

- Source-to-source compilers are a type of compiler that takes a high-level language as its input and outputs a high-level language. For example, an automatic parallelizing compiler will frequently take in a high-level language program as an input and then transform the code and annotate it with parallel code annotations (e.g. OpenMP) or language constructs (e.g. Fortran's `DOALL` statements).
- Bytecode compilers that compile to assembly language of a theoretical machine, like some Prolog implementations

    - This Prolog machine is also known as the Warren Abstract Machine (or WAM).
    - Bytecode compilers for Java, Python are also examples of this category.
- Just-in-time compilers (JIT compiler) defer compilation until runtime. JIT compilers exist for many modern languages including Python, JavaScript, Smalltalk, Java, Microsoft .NET's Common Intermediate Language (CIL) and others. A JIT compiler generally runs inside an interpreter. When the interpreter detects that a code path is "hot", meaning it is executed frequently, the JIT compiler will be invoked and compile the "hot" code for increased performance.
    - For some languages, such as Java, applications are first compiled using a bytecode compiler and delivered in a machine-independent intermediate representation. A bytecode interpreter executes the bytecode, but the JIT compiler will translate the bytecode to machine code when increased performance is necessary.[47]
- Hardware compilers (also known as syntheses tools) are compilers whose output is a description of the hardware configuration instead of a sequence of instructions.
    - The output of these compilers target computer hardware at a very low level, for example a field-programmable gate array (FPGA) or structured application-specific integrated circuit (ASIC).[48] Such compilers are said to be hardware compilers, because the source code they compile effectively controls the final configuration of the hardware and how it operates. The output of the compilation is only an interconnection of transistors or lookup tables.
    - An example of hardware compiler is XST, the Xilinx Synthesis Tool used for configuring FPGAs.[49] Similar tools are available from Altera,[50] Synplicity, Synopsys and other hardware vendors.
- An *assembler* is a program that compiles human readable assembly language to machine code, the actual instructions executed by hardware. The inverse program that translates machine code to assembly language is called a disassembler.
- A program that translates from a low-level language to a higher level one is a decompiler.
- A program that translates between high-level languages is usually called a language translator, source-to-source compiler, language converter, or language rewriter. The last term is usually applied to translations that do not involve a change of language.[51]
- A program that translates into an object code format that is not supported on the compilation machine is called a cross compiler and is commonly used to prepare code for embedded applications.
- A program that rewrites object code back into the same type of object code while applying optimisations and transformations is a binary recompiler.

See also

- Abstract interpretation
- Bottom-up parsing
- Compile and go loader
- Compile farm
- List of compilers
- List of important publications in computer science § Compilers