Background Material for The Engineering of Compilers and Machine Instruction Sets

Dr. Bart Stuck

18 November 2023

Source: Wikipedia.org

Outline

- Background Information
- History by Languages
  - Fortran and Basic and Lisp and Cobol
  - PL1
  - C and C++
  - Ada
  - Java
  - Python and Pearl

What Is A Compiler?

- In computing, a **compiler** is a computer program that translates computer code written in one programming language (the *source* language) into another language (the *target* language).
- The name "compiler" is primarily used for programs that translate source code from a high-level programming language to a low-level programming language (e.g. assembly language, object code, or machine code) to create an executable program.

Many Types of Compilers

- There are many different types of compilers which produce output in different useful forms.
  - A *cross-compiler* produces code for a different CPU or operating system than the one on which the cross-compiler itself runs.
  - A *bootstrap compiler* is often a temporary compiler, used for compiling a more permanent or better optimised compiler for a language.
- Related software include, 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, 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. A *compiler-compiler* is a compiler that produces a compiler (or part of one), often in a generic and reusable way so as to be able to produce many differing compilers.

Steps in Running a Compiler

- A compiler is likely to perform some or all of the following operations, often called phases:
  - preprocessing,
  - lexical analysis,
  - parsing,
  - semantic analysis (syntax-directed translation), conversion of input programs to an intermediate representation, code optimization and machine specific code generation.
- Compilers generally implement these phases as modular components, promoting efficient design and correctness of 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.

Compiler vs Interpreter

- 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.
- The translation process influences the design of computer languages, which leads to a preference of compilation or interpretation.
- In theory, a programming language can have both a compiler and an interpreter.
- In practice, programming languages tend to be associated with just one (a compiler or an interpreter).

High Level Language

- 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.
- Backus–Naur form (BNF) describes the syntax of "sentences" of a language and was used for the syntax of Algol 60 by John Backus. The ideas derive from the context-free grammar concepts by Noam Chomsky, a linguist.[7] "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."

Initial High Level Languages

- 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 one of the first actually implemented high-level languages and first optimizing compiler.
  - COBOL (Common Business-Oriented Language) evolved from A-0 and FLOW-MATIC to become the dominant high-level language for business applications.
  - LISP (List Processor) for symbolic computation.
- 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.

Early Milestones in 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),[17][18] 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.[*citation needed*]
- *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.
- *1958–1960*: Algol 58 was the precursor to ALGOL 60. Algol 58 introduced code blocks, a key advance in the rise of structured programming. ALGOL 60 was the first language to implement nested function definitions with lexical scope. It included recursion. Its syntax was defined using BNF. ALGOL 60 inspired many languages that followed it. Tony Hoare remarked: "... it was not only an improvement on its predecessors but also on nearly all its successors."[19][20]
- *1958–1962*: John McCarthy at MIT designed LISP.[21] 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.

Operating Systems Move to High Level Languages

- Early operating systems and software were written in assembly language. In the 1960s and early 1970s, 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. Several compilers have been implemented, Richards' book provides insights to the language and its compiler. BCPL was not only an influential systems programming language that is still used in research 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 and UNIX High Level Languages

- 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. Multics was written in the PL/I language developed by IBM and IBM User Group. 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. For the first few years of the Multics 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. EPL supported the project until a boot-strapping compiler for the full PL/I could be developed.
- Bell Labs left the Multics project in 1969, and developed a system programming language B based on BCPL concepts, 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.

Evolution of C and C++
Higher Level Languages

- Bell Labs started the 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.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.
- 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. Bell Labs became interested in OOP with the development of C++. 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.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 and Ada High Level Language

- 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. 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.[38] 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. 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 (since 1995, object-oriented) programming language Ada.

Ada Programming Language Evolution

- 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 overall effort on Ada development.

High Level Language Evolution Drives Compiler Technology Evolution

- 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.

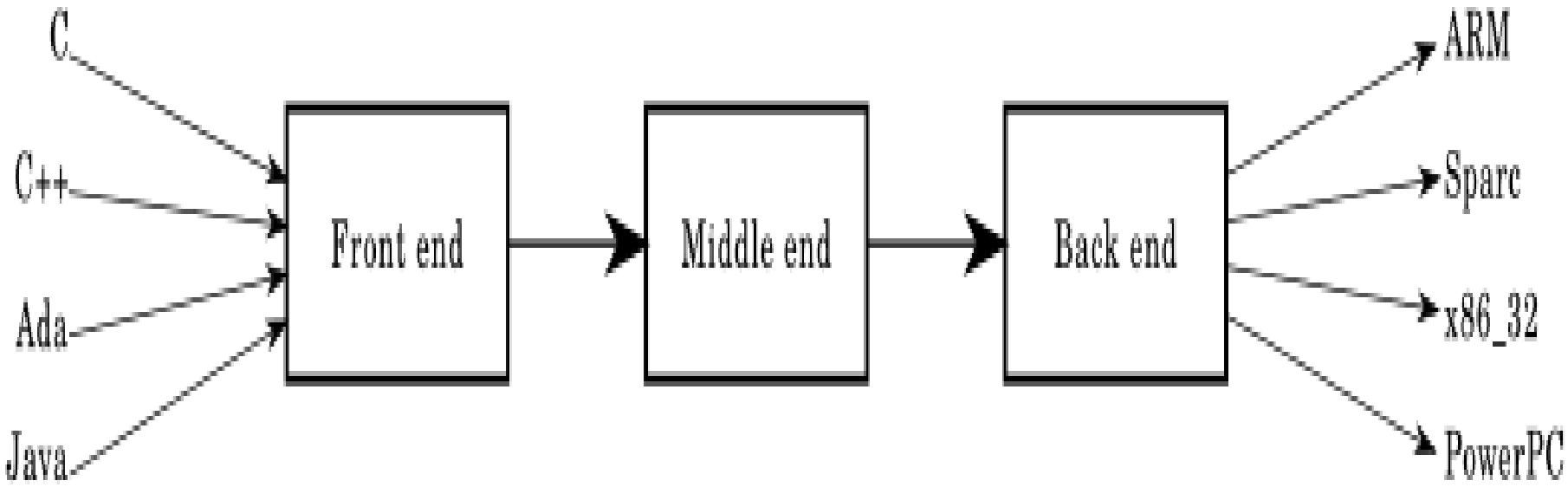
Compiler Design and Implementation

- 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 vs Multi-Pass Compilers

- Classifying compilers by number of passes has its background in the hardware resource limitations of computers. Compiling involves performing much 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.

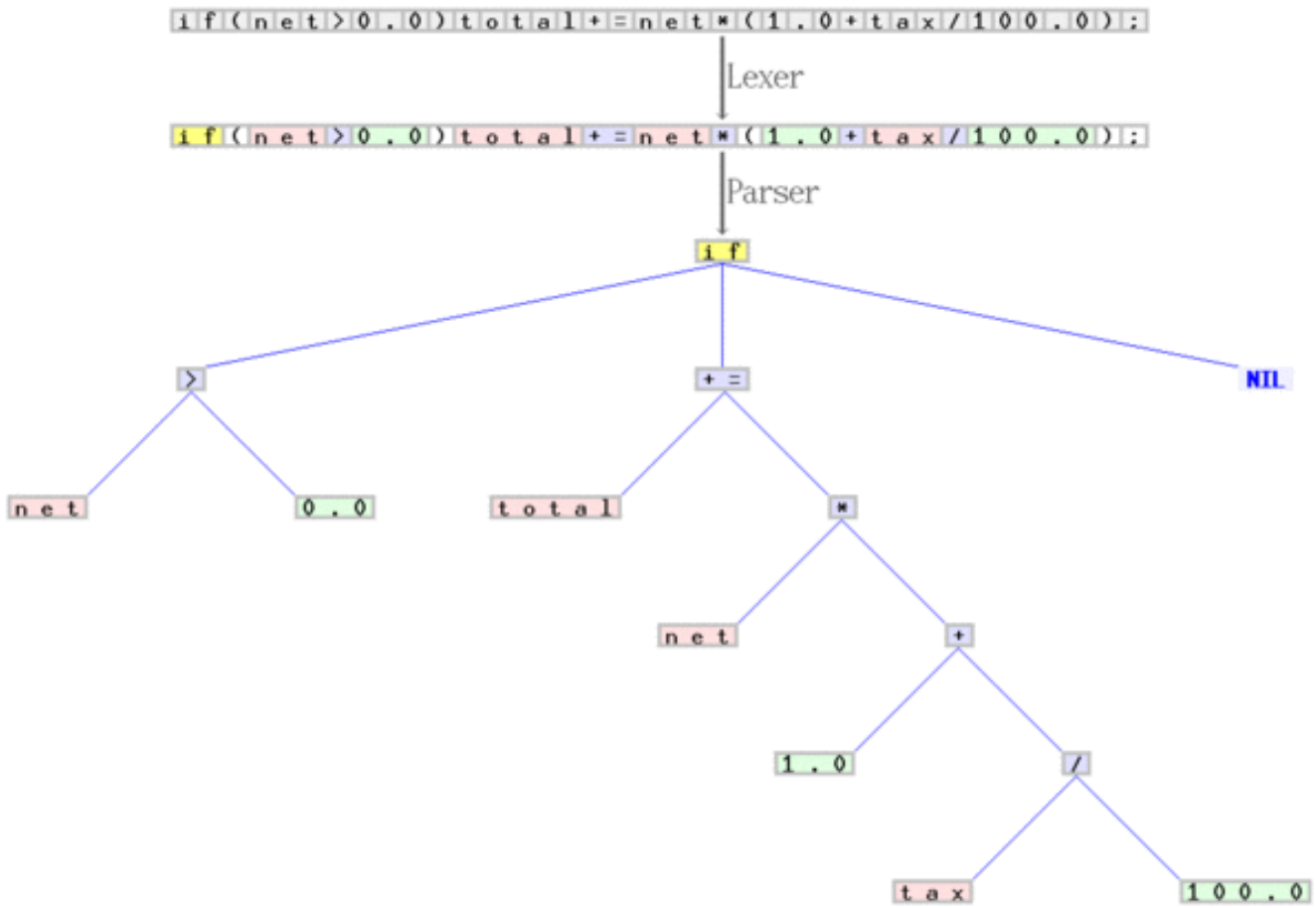
Three Stage Compiler Structure



Front End of Compiler

- The *front end* scans the input and 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 error and/or warning messages, usually identifying the location in the source code where the problem was detected; in some cases the actual error may be (much) earlier in the program.
- 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.

Front End Example: Lexer and Parser for C



Front End Activities

- *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*.[46] This phase can be divided into two stages: the *scanning*, which segments the input text into syntactic units called *lexemes* and assigns 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*.[47] 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.[48]
- *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 of Compiler Implementation

- The *middle* 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 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.

Middle Compiler Optimization

- The middle also known as *optimizer* performs optimizations on the intermediate representation in order to improve the performance and the quality of the produced machine code. 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.

Back End of Compiler Interpretation

- 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 optimization problems are NP-hard, heuristic techniques for solving them are well-developed and implemented in production-quality compilers. Typically the output of a back end is machine code specialized for a particular processor and operating system.

Back End Phases

- *Machine dependent optimizations*: optimizations that depend on the details of the CPU architecture that the compiler targets.[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.

Three Stage Compiler Pipeline

- 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.
- Practical examples of this approach are the GNU Compiler Collection, Clang (LLVM-based C/C++ compiler), and the Amsterdam Compiler Kit, which have multiple front-ends, shared optimizations and multiple back-ends.