Programming Languages in Computer Science

Josephine Pulley

First Edition, 2012

ISBN 978-81-323-3095-0

Published by:
Research World
4735/22 Prakashdeep Bldg,
Ansari Road, Darya Ganj,
Delhi - 110002
Email: info@wtbooks.com

Table of Contents

Chapter 1

Programming Language

A **programming language** is an artificial language designed to express computations that can be performed by a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine, to express algorithms precisely, or as a mode of human communication.

The earliest programming languages predate the invention of the computer, and were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, with many more being created every year. Most programming languages describe computation in an imperative style, i.e., as a sequence of commands, although some languages, such as those that support functional programming or logic programming, use alternative forms of description.

A programming language is usually split into the two components of syntax (form) and semantics (meaning) and many programming languages have some kind of written specification of their syntax and/or semantics. Some languages are defined by a specification document, for example, the C programming language is specified by an ISO Standard, while other languages, such as Perl, have a dominant implementation that is used as a reference.

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms. Traits often considered important for what constitutes a programming language include:

- *Function and target:* A *computer programming language* is a language used to write computer programs, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example PostScript programs are frequently

created by another program to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language. In most practical contexts, a programming language involves a computer; consequently programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

- *Abstractions:* Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.
- *Expressive power:* The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.

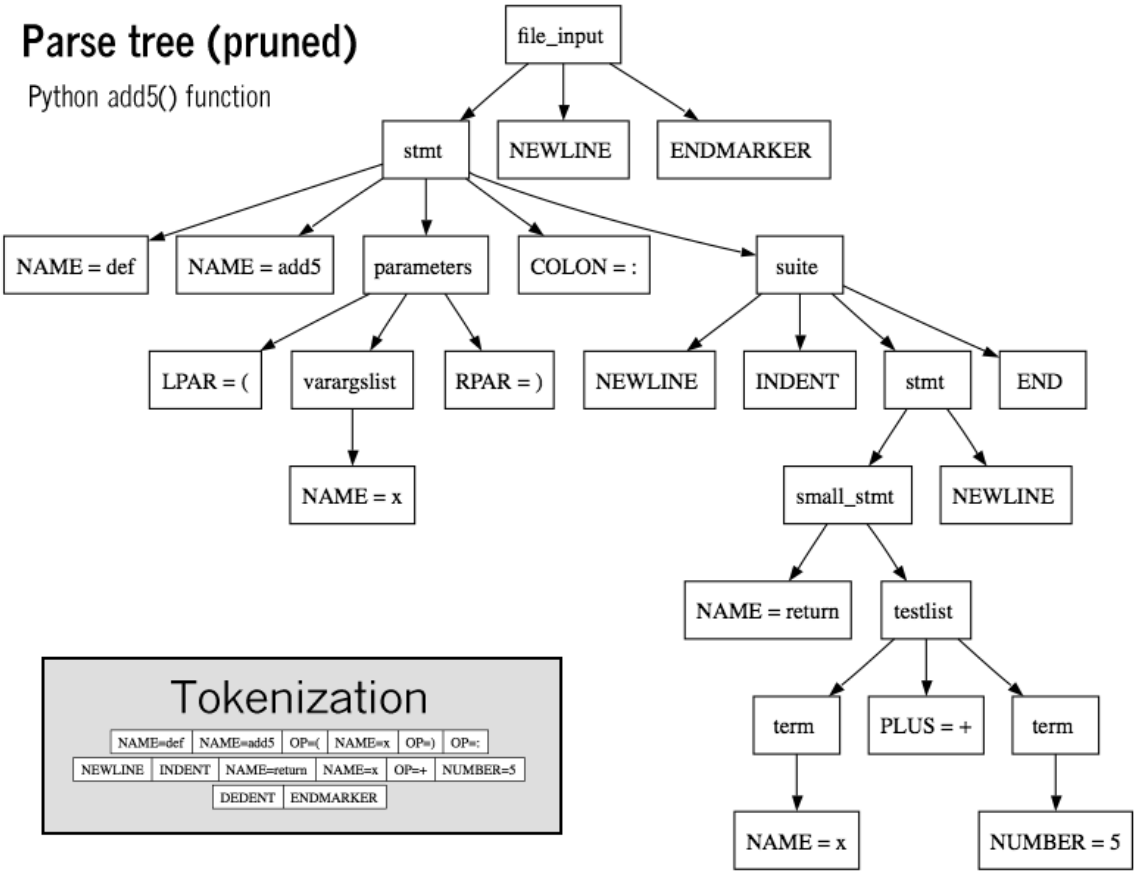
Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.

The term *computer language* is sometimes used interchangeably with programming language. However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages. In this vein, languages used in computing that have a different goal than expressing computer programs are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming. Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources. John C. Reynolds emphasizes that formal specification languages are just as much programming languages as are the languages intended for execution. He also argues that textual and even graphical input formats that affect the behavior of a computer are programming languages, despite the fact they are commonly not Turing-complete, and remarks that ignorance of programming language concepts is the reason for many flaws in input formats.

Elements

All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe their structure and meaning respectively.

Syntax



Parse tree of Python code with inset tokenization

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
    else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Syntax highlighting is often used to aid programmers in recognizing elements of source code. The language above is Python.

A programming language's surface form is known as its syntax. Most programming languages are purely textual; they use sequences of text including words, numbers, and punctuation, much like written natural languages. On the other hand, there are some programming languages which are more graphical in nature, using visual relationships between symbols to specify a program.

The syntax of a language describes the possible combinations of symbols that form a syntactically correct program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Since most languages are textual.

Programming language syntax is usually defined using a combination of regular expressions (for lexical structure) and Backus–Naur Form (for grammatical structure). Below is a simple grammar, based on Lisp:

```
expression ::= atom | list
atom ::= number | symbol
number ::= [+-]?['0'-'9']+
symbol ::= ['A'-'Z''a'-'z'].*
list ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

The following are examples of well-formed token sequences in this grammar: '`12345`', '`()`','`(a b c232 (1))`'

Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because `p` is a null pointer, the operations `p->real` and `p->im` have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

If the type declaration on the first line were omitted, the program would trigger an error on compilation, as the variable "p" would not be defined. But the program would still be syntactically correct, since type declarations provide only semantic information.

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars. Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution. In contrast to Lisp's macro system and Perl's `BEGIN`

blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.

Semantics

The term *semantics* refers to the meaning of languages, as opposed to their form (syntax).

Static semantics

The static semantics defines restrictions on the structure of valid texts that are hard or impossible to express in standard syntactic formalisms. For compiled languages, static semantics essentially include those semantic rules that can be checked at compile time. Examples include checking that every identifier is declared before it is used (in languages that require such declarations) or that the labels on the arms of a case statement are distinct. Many important restrictions of this type, like checking that identifiers are used in the appropriate context (e.g. not adding a integer to a function name), or that subroutine calls have the appropriate number and type of arguments can be enforced by defining them as rules in a logic called a type system. Other forms of static analyses like data flow analysis may also be part of static semantics. Newer programming languages like Java and C# have definite assignment analysis, a form of data flow analysis, as part of their static semantics.

Dynamic semantics

Once data has been specified, the machine must be instructed to perform operations on the data. For example, the semantics may define the strategy by which expressions are evaluated to values, or the manner in which control structures conditionally execute statements. The *dynamic semantics* (also known as *execution semantics*) of a language defines how and when the various constructs of a language should produce a program behavior. There are many ways of defining execution semantics. Natural language is often used to specify the execution semantics of languages commonly used in practice. A significant amount of academic research went into formal semantics of programming languages, which allow execution semantics to be specified in a formal manner. Results from this field of research have seen limited application to programming language design and implementation outside academia.

Type system

A type system defines how a programming language classifies values and expressions into *types*, how it can manipulate those types and how they interact. The goal of a type system is to verify and usually enforce a certain level of correctness in programs written in that language by detecting certain incorrect operations. Any decidable type system involves a trade-off: while it rejects many incorrect programs, it can also prohibit some correct, albeit unusual programs. In order to bypass this downside, a number of languages have *type loopholes*, usually unchecked casts that may be used by the programmer to explicitly allow a normally disallowed operation between different types. In most typed

languages, the type system is used only to type check programs, but a number of languages, usually functional ones, infer types, relieving the programmer from the need to write type annotations. The formal design and study of type systems is known as *type theory*.

Typed versus untyped languages

A language is *typed* if the specification of every operation defines types of data to which the operation is applicable, with the implication that it is not applicable to other types. For example, the data represented by "`this text between the quotes`" is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any program attempting to perform such an operation. In some languages, the meaningless operation will be detected when the program is compiled ("static" type checking), and rejected by the compiler, while in others, it will be detected when the program is run ("dynamic" type checking), resulting in a runtime exception.

A special case of typed languages are the *single-type* languages. These are often scripting or markup languages, such as REXX or SGML, and have only one data type—most commonly character strings which are used for both symbolic and numeric data.

In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths. High-level languages which are untyped include BCPL and some varieties of Forth.

In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing. Many production languages provide means to bypass or subvert the type system.

Static versus dynamic typing

In *static typing* all expressions have their types determined prior to the program being run (typically at compile-time). For example, 1 and (2+2) are integer expressions; they cannot be passed to a function that expects a string, or stored in a variable that is defined to hold dates.

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.

Dynamic typing, also called *latent typing*, determines the type-safety of operations at runtime; in other words, types are associated with *runtime values* rather than *textual expressions*. As with type-inferred languages, dynamically typed languages do not require the programmer to write explicit type annotations on expressions. Among other things, this may permit a single variable to refer to values of different types at different points in the program execution. However, type errors cannot be automatically detected until a piece of code is actually executed, potentially making debugging more difficult. Ruby, Lisp, JavaScript, and Python are dynamically typed.

Weak and strong typing

Weak typing allows a value of one type to be treated as another, for example treating a string as a number. This can occasionally be useful, but it can also allow some kinds of program faults to go undetected at compile time and even at runtime.

Strong typing prevents the above. An attempt to perform an operation on the wrong type of value raises an error. Strongly typed languages are often termed *type-safe* or *safe*.

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression `2 * x` implicitly converts `x` to a number, and this conversion succeeds even if `x` is `null`, `undefined`, an `Array`, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors. *Strong* and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

Standard library and run-time system

Most programming languages have an associated core library (sometimes known as the 'standard library', especially if it is included as part of the published language standard), which is conventionally made available by all implementations of the language. Core libraries typically include definitions for commonly used algorithms, data structures, and mechanisms for input and output.

A language's core library is often treated as part of the language by its users, although the designers may have treated it as a separate entity. Many language specifications define a core that must be made available in all implementations, and in the case of standardized languages this core library may be required. The line between a language and its core library therefore differs from language to language. Indeed, some languages are designed so that the meanings of certain syntactic constructs cannot even be described without referring to the core library. For example, in Java, a string literal is defined as an instance of the `java.lang.String` class; similarly, in Smalltalk, an anonymous function expression (a "block") constructs an instance of the library's `BlockContext` class. Conversely, Scheme contains multiple coherent subsets that suffice to construct the rest

of the language as library macros, and so the language designers do not even bother to say which portions of the language must be implemented as language constructs, and which must be implemented as parts of a library.

Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another. But as artificial constructs, they also differ in fundamental ways from languages that have evolved through usage. A significant difference is that a programming language can be fully described and studied in its entirety, since it has a precise and finite definition. By contrast, natural languages have changing meanings given by their users in different communities. While constructed languages are also artificial languages designed from the ground up with a specific purpose, they lack the precise and complete semantic definition that a programming language has.

Many languages have been designed from scratch, altered to meet new needs, combined with other languages, and eventually fallen into disuse. Although there have been attempts to design one "universal" programming language that serves all purposes, all of them have failed to be generally accepted as filling this role. The need for diverse programming languages arises from the diversity of contexts in which languages are used:

- Programs range from tiny scripts written by individual hobbyists to huge systems written by hundreds of programmers.
- Programmers range in expertise from novices who need simplicity above all else, to experts who may be comfortable with considerable complexity.
- Programs must balance speed, size, and simplicity on systems ranging from microcontrollers to supercomputers.
- Programs may be written once and not change for generations, or they may undergo continual modification.
- Finally, programmers may simply differ in their tastes: they may be accustomed to discussing problems and expressing them in a particular language.

One common trend in the development of programming languages has been to add more ability to solve problems using a higher level of abstraction. The earliest programming languages were tied very closely to the underlying hardware of the computer. As new programming languages have developed, features have been added that let programmers express ideas that are more remote from simple translation into underlying hardware instructions. Because programmers are less tied to the complexity of the computer, their programs can do more computing with less effort from the programmer. This lets them write more functionality per time unit.

Natural language processors have been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its

benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish". Alan Perlis was similarly dismissive of the idea. Hybrid approaches have been taken in Structured English and SQL.

A language's designers and users must construct a number of artifacts that govern and enable the practice of programming. The most important of these artifacts are the language *specification* and *implementation*.

Specification

The **specification** of a programming language is intended to provide a definition that the language users and the implementors can use to determine whether the behavior of a program is correct, given its source code.

A programming language specification can take several forms, including the following:

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML and Scheme specifications).
- A description of the behavior of a translator for the language (e.g., the C++ and Fortran specifications). The syntax and semantics of the language have to be inferred from this description, which may be written in natural or a formal language.
- A *reference* or *model* implementation, sometimes written in the language being specified (e.g., Prolog or ANSI REXX). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

An **implementation** of a programming language provides a way to execute that program on one or more configurations of hardware and software. There are, broadly, two approaches to programming language implementation: *compilation* and *interpretation*. It is generally possible to implement a language using either technique.

The output of a compiler may be executed by hardware or a program called an interpreter. In some implementations that make use of the interpreter approach there is no distinct boundary between compiling and interpreting. For instance, some implementations of BASIC compile and then execute the source a line at a time.

Programs that are executed directly on the hardware usually run several orders of magnitude faster than those that are interpreted in software.

One technique for improving the performance of interpreted programs is just-in-time compilation. Here the virtual machine, just before execution, translates the blocks of bytecode which are going to be used to machine code, for direct execution on the hardware.

Usage

Thousands of different programming languages have been created, mainly in the computing field. Programming languages differ from most other forms of human expression in that they require a greater degree of precision and completeness. When using a natural language to communicate with other people, human authors and speakers can be ambiguous and make small errors, and still expect their intent to be understood. However, figuratively speaking, computers "do exactly what they are told to do", and cannot "understand" what code the programmer intended to write. The combination of the language definition, a program, and the program's inputs must fully specify the external behavior that occurs when the program is executed, within the domain of control of that program.

A programming language provides a structured mechanism for defining pieces of data, and the operations or transformations that may be carried out automatically on that data. A programmer uses the abstractions present in the language to represent the concepts involved in a computation. These concepts are represented as a collection of the simplest elements available (called primitives). *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

Programs for a computer might be executed in a batch process without human interaction, or a user might type commands in an interactive session of an interpreter. In this case the "commands" are simply programs, whose execution is chained together. When a language is used to give commands to a software application (such as a shell) it is called a scripting language.

Measuring language usage

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; FORTRAN in scientific and engineering applications; C in embedded applications and operating systems; and other languages are regularly used to write many different kinds of applications.

Various methods of measuring language popularity, each subject to a different bias over what is measured, have been proposed:

- counting the number of job advertisements that mention the language
- the number of books sold that teach or describe the language
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

Taxonomies

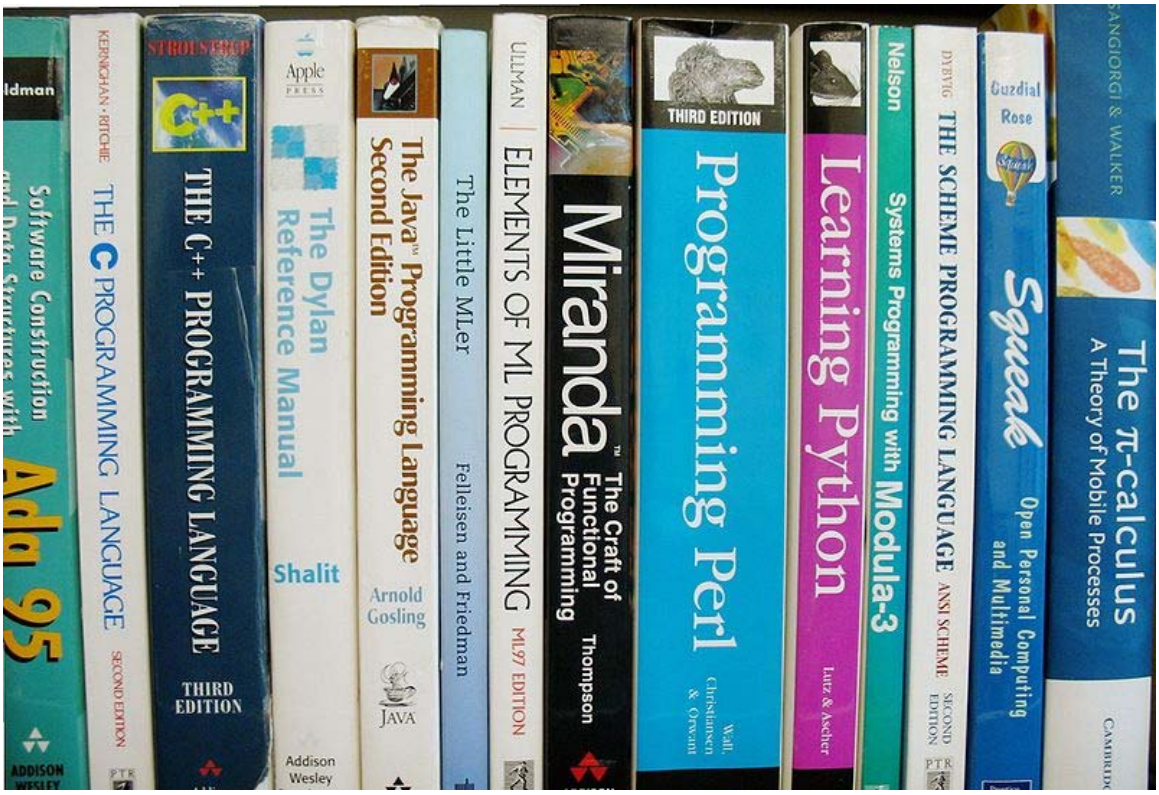
There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in one language will diffuse throughout a family of related languages, and then leap suddenly across familial gaps to appear in an entirely different family.

The task is further complicated by the fact that languages can be classified along multiple axes. For example, Java is both an object-oriented language (because it encourages object-oriented organization) and a concurrent language (because it contains built-in constructs for running multiple threads in parallel). Python is an object-oriented scripting language.

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use*. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming. More refined paradigms include procedural programming, object-oriented programming, functional programming, and logic programming; some languages are hybrids of paradigms or multi-paradigmatic. An assembly language is not so much a paradigm as a direct model of an underlying machine architecture. By purpose, programming languages might be considered general purpose, system programming languages, scripting languages, domain-specific languages, or concurrent/distributed languages (or a combination of these). Some general purpose languages were designed largely with educational goals.

A programming language may also be classified by factors unrelated to programming paradigm. For instance, most programming languages use English language keywords, while a minority do not. Other languages may be classified as being esoteric or not.

History



A selection of textbooks that teach programming, in languages both popular and obscure. These are only a few of the thousands of programming languages and dialects that have been designed in history.

Early developments

The first programming languages predate the modern computer. The 19th century had "programmable" looms and player piano scrolls which implemented what are today recognized as examples of domain-specific languages. By the beginning of the twentieth century, punch cards encoded data and directed mechanical processing. In the 1930s and 1940s, the formalisms of Alonzo Church's lambda calculus and Alan Turing's Turing machines provided mathematical abstractions for expressing algorithms; the lambda calculus remains influential in language design.

In the 1940s, the first electrically powered digital computers were created. The first high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.

Programmers of early 1950s computers, notably UNIVAC I and IBM 701, used machine language programs, that is, the first generation language (1GL). 1GL programming was quickly superseded by similarly machine-specific, but mnemonic, second generation languages (2GL) known as assembly languages or "assembler". Later in the 1950s,

assembly language programming, which had evolved to include the use of macro instructions, was followed by the development of "third generation" programming languages (3GL), such as FORTRAN, LISP, and COBOL. 3GLs are more abstract and are "portable", or at least implemented similarly on computers that do not support the same native machine code. Updated versions of all of these 3GLs are still in general use, and each has strongly influenced the development of later languages. At the end of the 1950s, the language formalized as ALGOL 60 was introduced, and most later programming languages are, in many respects, descendants of Algol. The format and use of the early programming languages was heavily influenced by the constraints of the interface.

Refinement

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use, though many aspects were refinements of ideas in the very first Third-generation programming languages:

- APL introduced *array programming* and influenced functional programming.
- PL/I (NPL) was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.
- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.
- Prolog, designed in 1972, was the first *logic programming* language.
- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it. Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.

The 1960s and 1970s also saw expansion of techniques that reduced the footprint of a program as well as improved productivity of the programmer and user. The card deck for an early 4GL was a lot smaller for the same functionality expressed in a 3GL deck.

Consolidation and growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems

programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.

The rapid growth of the Internet in the mid-1990s created opportunities for new languages. Perl, originally a Unix scripting tool first released in 1987, became common in dynamic websites. Java came to be used for server-side programming, and bytecode virtual machines became popular again in commercial settings with their promise of "Write once, run anywhere" (UCSD Pascal had been popular for a time in the early 1980s). These developments were not fundamentally novel, rather they were refinements to existing languages and paradigms, and largely based on the C family of programming languages.

Programming language evolution continues, in both industry and research. Current directions include security and reliability verification, new kinds of modularity (mixins, delegates, aspects), and database integration such as Microsoft's LINQ.

The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its 'here document' can hold multiple 4GL programs, as well as multiple JavaScript programs, in part of its own perl code and use variable interpolation in the 'here document' to support multi-language programming.

Chapter 2

Computer Programming

Computer programming (often shortened to **programming** or **coding**) is the process of designing, writing, testing, debugging / troubleshooting, and maintaining the source code of computer programs. This source code is written in a programming language. The purpose of programming is to create a program that exhibits a certain desired behaviour. The process of writing source code often requires expertise in many different subjects, including knowledge of the application domain, specialized algorithms and formal logic.

Definition

Hoc and Nguyen-Xuan define computer programming as "the process of transforming a mental plan in familiar terms into one compatible with the computer." Said another way, programming is the craft of transforming requirements into something that a computer can execute.

Overview

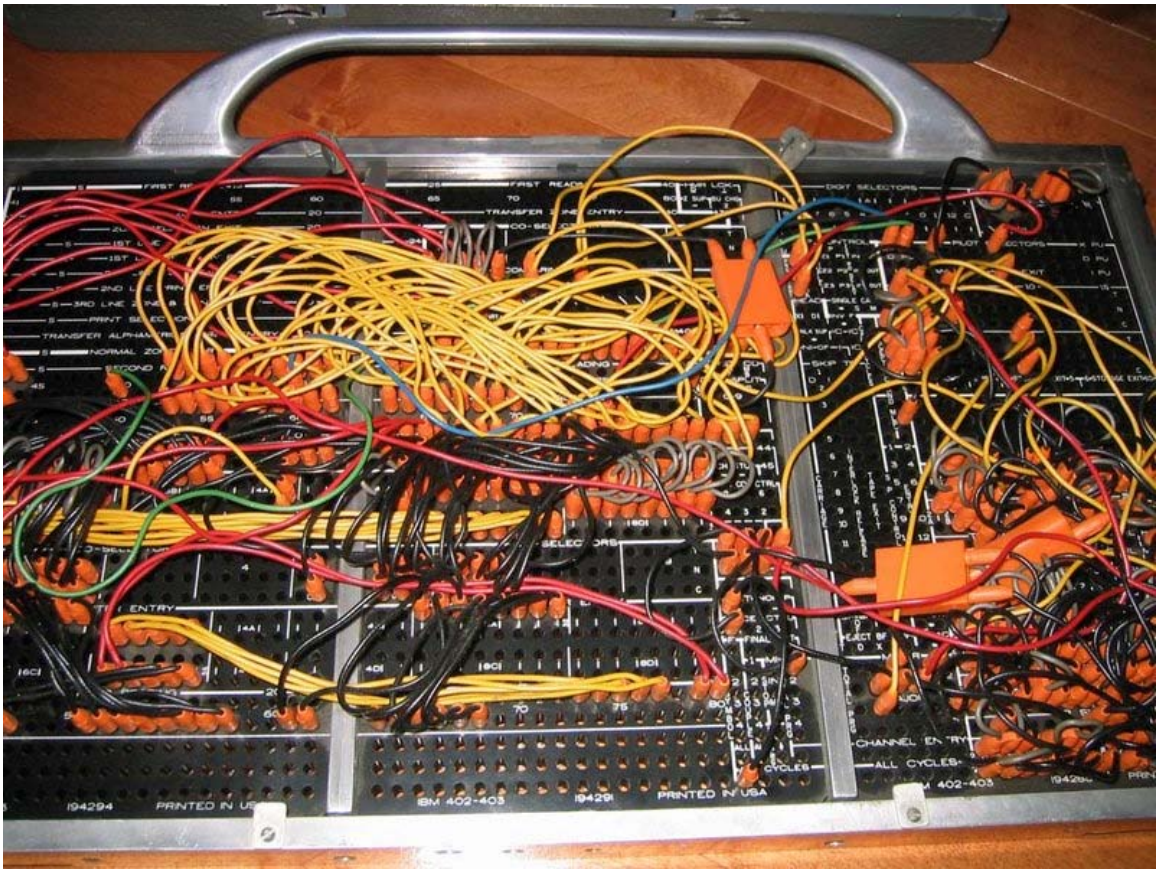
Within software engineering, programming (the *implementation*) is regarded as one phase in a software development process.

There is an ongoing debate on the extent to which the writing of programs is an art, a craft or an engineering discipline. In general, good programming is considered to be the measured application of all three, with the goal of producing an efficient and evolvable software solution (the criteria for "efficient" and "evolvable" vary considerably). The discipline differs from many other technical professions in that programmers, in general, do not need to be licensed or pass any standardized (or governmentally regulated) certification tests in order to call themselves "programmers" or even "software engineers." However, representing oneself as a "Professional Software Engineer" without a license from an accredited institution is illegal in many parts of the world. However, because the discipline covers many areas, which may or may not include critical applications, it is debatable whether licensing is required for the profession as a whole. In most cases, the discipline is self-governed by the entities which require the programming,

and sometimes very strict environments are defined (e.g. United States Air Force use of AdaCore and security clearance).

Another ongoing debate is the extent to which the programming language used in writing computer programs affects the form that the final program takes. This debate is analogous to that surrounding the Sapir–Whorf hypothesis in linguistics, which postulates that a particular spoken language's nature influences the habitual thought of its speakers. Different language patterns yield different patterns of thought. This idea challenges the possibility of representing the world perfectly with language, because it acknowledges that the mechanisms of any language condition the thoughts of its speaker community.

History

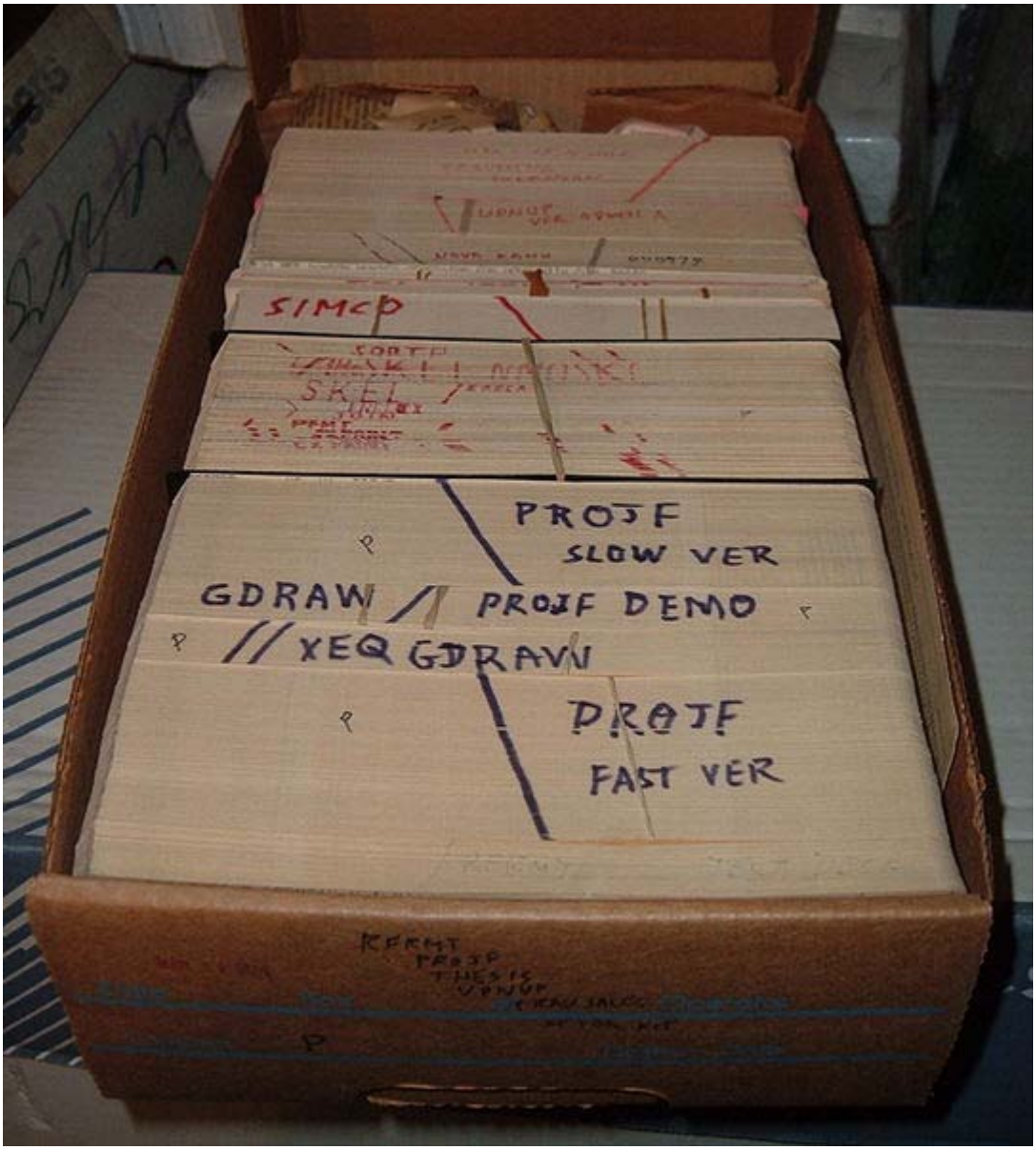


Wired plug board for an IBM 402 Accounting Machine.

The Antikythera mechanism from ancient Greece was a calculator utilizing gears of various sizes and configuration to determine its operation, which tracked the metonic cycle still used in lunar-to-solar calendars, and which is consistent for calculating the dates of the Olympiads. Al-Jazari built programmable Automata in 1206. One system employed in these devices was the use of pegs and cams placed into a wooden drum at specific locations. which would sequentially trigger levers that in turn operated percussion instruments. The output of this device was a small drummer playing various rhythms and drum patterns. The Jacquard Loom, which Joseph Marie Jacquard developed

in 1801, uses a series of pasteboard cards with holes punched in them. The hole pattern represented the pattern that the loom had to follow in weaving cloth. The loom could produce entirely different weaves using different sets of cards. Charles Babbage adopted the use of punched cards around 1830 to control his Analytical Engine. The synthesis of numerical calculation, predetermined operation and output, along with a way to organize and input instructions in a manner relatively easy for humans to conceive and produce, led to the modern development of computer programming. Development of computer programming accelerated through the Industrial Revolution.

In the late 1880s, Herman Hollerith invented the recording of data on a medium that could then be read by a machine. Prior uses of machine readable media, above, had been for control, not data. "After some initial trials with paper tape, he settled on punched cards..." To process these punched cards, first known as "Hollerith cards" he invented the tabulator, and the keypunch machines. These three inventions were the foundation of the modern information processing industry. In 1896 he founded the *Tabulating Machine Company* (which later became the core of IBM). The addition of a control panel (plugboard) to his 1906 Type I Tabulator allowed it to do different jobs without having to be physically rebuilt. By the late 1940s, there were a variety of plug-board programmable machines, called unit record equipment, to perform data-processing tasks (card reading). Early computer programmers used plug-boards for the variety of complex calculations requested of the newly invented machines.



Data and instructions could be stored on external punched cards, which were kept in order and arranged in program decks.

The invention of the von Neumann architecture allowed computer programs to be stored in computer memory. Early programs had to be painstakingly crafted using the instructions (elementary operations) of the particular machine, often in binary notation. Every model of computer would likely use different instructions (machine language) to do the same task. Later, assembly languages were developed that let the programmer specify each instruction in a text format, entering abbreviations for each operation code instead of a number and specifying addresses in symbolic form (e.g., ADD X, TOTAL). Entering a program in assembly language is usually more convenient, faster, and less prone to human error than using machine language, but because an assembly language is

little more than a different notation for a machine language, any two machines with different instruction sets also have different assembly languages.

In 1954, FORTRAN was invented; it was the first high level programming language to have a functional implementation, as opposed to just a design on paper. (A high-level language is, in very general terms, any programming language that allows the programmer to write programs in terms that are more abstract than assembly language instructions, i.e. at a level of abstraction "higher" than that of an assembly language.) It allowed programmers to specify calculations by entering a formula directly (e.g. Y = X*2 + 5*X + 9). The program text, or *source*, is converted into machine instructions using a special program called a compiler, which translates the FORTRAN program into machine language. In fact, the name FORTRAN stands for "Formula Translation". Many other languages were developed, including some for commercial programming, such as COBOL. Programs were mostly still entered using punched cards or paper tape. By the late 1960s, data storage devices and computer terminals became inexpensive enough that programs could be created by typing directly into the computers. Text editors were developed that allowed changes and corrections to be made much more easily than with punched cards. (Usually, an error in punching a card meant that the card had to be discarded and a new one punched to replace it.)

As time has progressed, computers have made giant leaps in the area of processing power. This has brought about newer programming languages that are more abstracted from the underlying hardware. Although these high-level languages usually incur greater overhead, the increase in speed of modern computers has made the use of these languages much more practical than in the past. These increasingly abstracted languages typically are easier to learn and allow the programmer to develop applications much more efficiently and with less source code. However, high-level languages are still impractical for a few programs, such as those where low-level hardware control is necessary or where maximum processing speed is vital.

Throughout the second half of the twentieth century, programming was an attractive career in most developed countries. Some forms of programming have been increasingly subject to offshore outsourcing (importing software and services from other countries, usually at a lower wage), making programming career decisions in developed countries more complicated, while increasing economic opportunities in less developed areas. It is unclear how far this trend will continue and how deeply it will impact programmer wages and opportunities.

Modern programming

Quality requirements

Whatever the approach to software development may be, the final program must satisfy some fundamental properties. The following properties are among the most relevant:

- **Efficiency/performance**: the amount of system resources a program consumes (processor time, memory space, slow devices such as disks, network bandwidth and to some extent even user interaction): the less, the better. This also includes correct disposal of some resources, such as cleaning up temporary files and lack of memory leaks.
- **Reliability**: how often the results of a program are correct. This depends on conceptual correctness of algorithms, and minimization of programming mistakes, such as mistakes in resource management (e.g., buffer overflows and race conditions) and logic errors (such as division by zero or off-by-one errors).
- **Robustness**: how well a program anticipates problems not due to programmer error. This includes situations such as incorrect, inappropriate or corrupt data, unavailability of needed resources such as memory, operating system services and network connections, and user error.
- **Usability**: the ergonomics of a program: the ease with which a person can use the program for its intended purpose, or in some cases even unanticipated purposes. Such issues can make or break its success even regardless of other issues. This involves a wide range of textual, graphical and sometimes hardware elements that improve the clarity, intuitiveness, cohesiveness and completeness of a program's user interface.
- **Portability**: the range of computer hardware and operating system platforms on which the source code of a program can be compiled/interpreted and run. This depends on differences in the programming facilities provided by the different platforms, including hardware and operating system resources, expected behaviour of the hardware and operating system, and availability of platform specific compilers (and sometimes libraries) for the language of the source code.
- **Maintainability**: the ease with which a program can be modified by its present or future developers in order to make improvements or customizations, fix bugs and security holes, or adapt it to new environments. Good practices during initial development make the difference in this regard. This quality may not be directly apparent to the end user but it can significantly affect the fate of a program over the long term.

Readability of source code

In computer programming, readability refers to the ease with which a human reader can comprehend the purpose, control flow, and operation of source code. It affects the aspects of quality above, including portability, usability and most importantly maintainability.

Readability is important because programmers spend the majority of their time reading, trying to understand and modifying existing source code, rather than writing new source code. Unreadable code often leads to bugs, inefficiencies, and duplicated code. A study found that a few simple readability transformations made code shorter and drastically reduced the time to understand it.

Following a consistent programming style often helps readability. However, readability is more than just programming style. Many factors, having little or nothing to do with the

ability of the computer to efficiently compile and execute the code, contribute to readability. Some of these factors include:

- Different indentation styles (whitespace)
- Comments
- Decomposition
- Naming conventions for objects (such as variables, classes, procedures, etc)

Algorithmic complexity

The academic field and the engineering practice of computer programming are both largely concerned with discovering and implementing the most efficient algorithms for a given class of problem. For this purpose, algorithms are classified into *orders* using so-called Big O notation, *O(n)*, which expresses resource use, such as execution time or memory consumption, in terms of the size of an input. Expert programmers are familiar with a variety of well-established algorithms and their respective complexities and use this knowledge to choose algorithms that are best suited to the circumstances.

Methodologies

The first step in most formal software development projects is requirements analysis, followed by testing to determine value modeling, implementation, and failure elimination (debugging). There exist a lot of differing approaches for each of those tasks. One approach popular for requirements analysis is Use Case analysis. Nowadays many programmers use forms of Agile software development where the various stages of formal software development are more integrated together into short cycles that take a few weeks rather than years. There are many approaches to the Software development process.

Popular modeling techniques include Object-Oriented Analysis and Design (OOAD) and Model-Driven Architecture (MDA). The Unified Modeling Language (UML) is a notation used for both the OOAD and MDA.

A similar technique used for database design is Entity-Relationship Modeling (ER Modeling).

Implementation techniques include imperative languages (object-oriented or procedural), functional languages, and logic languages.

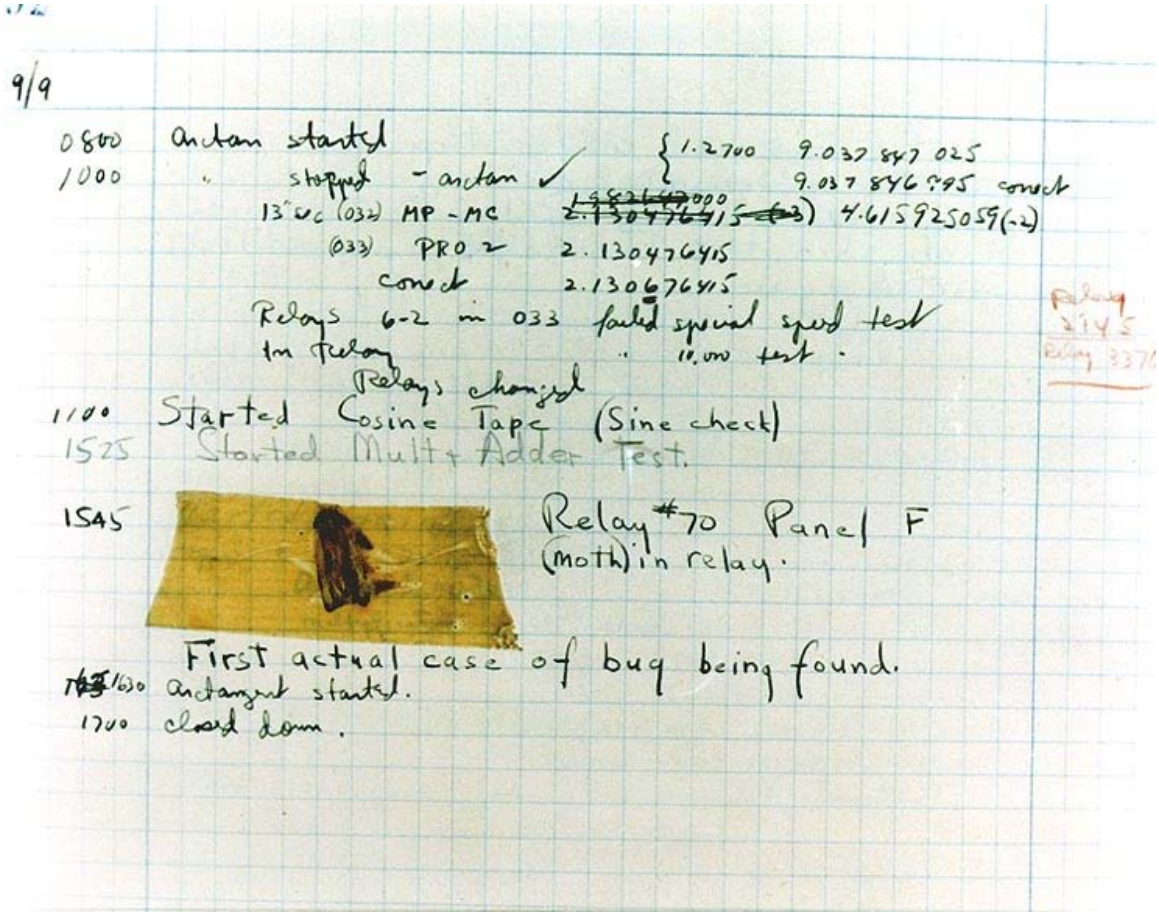
Measuring language usage

It is very difficult to determine what are the most popular of modern programming languages. Some languages are very popular for particular kinds of applications (e.g., COBOL is still strong in the corporate data center, often on large mainframes, FORTRAN in engineering applications, scripting languages in web development, and C in embedded applications), while some languages are regularly used to write many

different kinds of applications. Also many applications use a mix of several languages in their construction and use.

Methods of measuring programming language popularity include: counting the number of job advertisements that mention the language, the number of books teaching the language that are sold (this overestimates the importance of newer languages), and estimates of the number of existing lines of code written in the language (this underestimates the number of users of business languages such as COBOL).

Debugging



A bug, which was debugged in 1947

Debugging is a very important task in the software development process, because an incorrect program can have significant consequences for its users. Some languages are more prone to some kinds of faults because their specification does not require compilers to perform as much checking as other languages. Use of a static analysis tool can help detect some possible problems.

Debugging is often done with IDEs like Eclipse, Kdevelop, NetBeans, Code::Blocks, and Visual Studio. Standalone debuggers like gdb are also used, and these often provide less of a visual environment, usually using a command line.

Programming languages

Different programming languages support different styles of programming (called *programming paradigms*). The choice of language used is subject to many considerations, such as company policy, suitability to task, availability of third-party packages, or individual preference. Ideally, the programming language best suited for the task at hand will be selected. Trade-offs from this ideal involve finding enough programmers who know the language to build a team, the availability of compilers for that language, and the efficiency with which programs written in a given language execute. Languages form an approximate spectrum from "low-level" to "high-level"; "low-level" languages are typically more machine-oriented and faster to execute, whereas "high-level" languages are more abstract and easier to use but execute less quickly. It is usually easier to code in "high-level" languages than in "low-level" ones.

Allen Downey, in his book *How To Think Like A Computer Scientist*, writes:

> The details look different in different languages, but a few basic instructions appear in just about every language:
>
> - **input**: Get data from the keyboard, a file, or some other device.
> - **output**: Display data on the screen or send data to a file or other device.
> - **arithmetic**: Perform basic arithmetical operations like addition and multiplication.
> - **conditional execution**: Check for certain conditions and execute the appropriate sequence of statements.
> - **repetition**: Perform some action repeatedly, usually with some variation.

Many computer languages provide a mechanism to call functions provided by libraries such as in .dlls. Provided the functions in a library follow the appropriate run time conventions (e.g., method of passing arguments), then these functions may be written in any other language.

Programmers

Computer programmers are those who write computer software. Their jobs usually involve:

- Coding
- Compilation
- Debugging
- Documentation
- Integration

- Maintenance
- Requirements analysis
- Software architecture
- Software testing
- Specification

Chapter 3

Computer Program

A **computer program** (also a **software program**, or just a **program**) is a sequence of instructions written to perform a specified task for a computer. A computer requires programs to function, typically executing the program's instructions in a central processor. The program has an executable form that the computer can use directly to execute the instructions. The same program in its human-readable source code form, from which executable programs are derived (e.g., compiled), enables a programmer to study and develop its algorithms.

Computer source code is often written by computer programmers. Source code is written in a programming language that usually follows one of two main paradigms: imperative or declarative programming. Source code may be converted into an executable file (sometimes called an executable program or a binary) by a compiler and later executed by a central processing unit. Alternatively, computer programs may be executed with the aid of an interpreter, or may be embedded directly into hardware.

Computer programs may be categorized along functional lines: system software and application software. Two or more computer programs may run simultaneously on one computer, a process known as multitasking.

Programming

```
#include <stdio.h>

int main()
{

printf("Hello world!\n");
return 0;

}
```

Source code of a program written in the C programming language

Computer programming is the iterative process of writing or editing source code. Editing source code involves testing, analyzing, and refining, and sometimes coordinating with other programmers on a jointly developed program. A person who practices this skill is referred to as a computer programmer, software developer or coder. The sometimes lengthy process of computer programming is usually referred to as software development. The term software engineering is becoming popular as the process is seen as an engineering discipline.

Paradigms

Computer programs can be categorized by the programming language paradigm used to produce them. Two of the main paradigms are imperative and declarative.

Programs written using an imperative language specify an algorithm using declarations, expressions, and statements. A declaration couples a variable name to a datatype. For example: `var x: integer;` . An expression yields a value. For example: `2 + 2` yields 4. Finally, a statement might assign an expression to a variable or use the value of a variable to alter the program's control flow. For example: `x := 2 + 2; if x = 4 then do_something();` One criticism of imperative languages is the side effect of an assignment statement on a class of variables called non-local variables.

Programs written using a declarative language specify the properties that have to be met by the output. They do not specify details expressed in terms of the control flow of the executing machine but of the mathematical relations between the declared objects and their properties. Two broad categories of declarative languages are functional languages and logical languages. The principle behind functional languages (like Haskell) is to not allow side effects, which makes it easier to reason about programs like mathematical functions. The principle behind logical languages (like Prolog) is to define the problem to be solved — the goal — and leave the detailed solution to the Prolog system itself. The goal is defined by providing a list of subgoals. Then each subgoal is defined by further providing a list of its subgoals, etc. If a path of subgoals fails to find a solution, then that subgoal is backtracked and another path is systematically attempted.

The form in which a program is created may be textual or visual. In a visual language program, elements are graphically manipulated rather than textually specified.

Compiling or interpreting

A *computer program* in the form of a human-readable, computer programming language is called source code. Source code may be converted into an executable image by a compiler or executed immediately with the aid of an interpreter.

Either compiled or interpreted programs might be executed in a batch process without human interaction, but interpreted programs allow a user to type commands in an

interactive session. In this case the programs are the separate commands, whose execution occurs sequentially, and thus together. When a language is used to give commands to a software application (such as a shell) it is called a scripting language.

Compiled computer programs are commonly referred to as executables, binary images, or simply as binaries — a reference to the binary file format used to store the executable code. Compilers are used to translate source code from a programming language into either object code or machine code. Object code needs further processing to become machine code, and machine code is the central processing unit's native code, ready for execution.

Interpreted computer programs -in a batch or interactive session- are either decoded and then immediately executed or are decoded into some efficient intermediate representation for future execution. BASIC, Perl, and Python are examples of immediately executed computer programs. Alternatively, Java computer programs are compiled ahead of time and stored as a machine independent code called bytecode. Bytecode is then executed on request by an interpreter called a virtual machine.

The main disadvantage of interpreters is that computer programs run slower than when compiled. Interpreting code is slower than running the compiled version because the interpreter must decode each statement each time it is loaded and then perform the desired action. However, software development may be faster using an interpreter because testing is immediate when the compiling step is omitted. Another disadvantage of interpreters is that at least one must be present on the computer during computer program execution. By contrast, compiled computer programs need no compiler present during execution.

No properties of a programming language require it to be exclusively compiled or exclusively interpreted. The categorization usually reflects the most popular method of language execution. For example, BASIC is thought of as an interpreted language and C a compiled language, despite the existence of BASIC compilers and C interpreters. Some systems use just-in-time compilation (JIT) whereby sections of the source are compiled 'on the fly' and stored for subsequent executions.

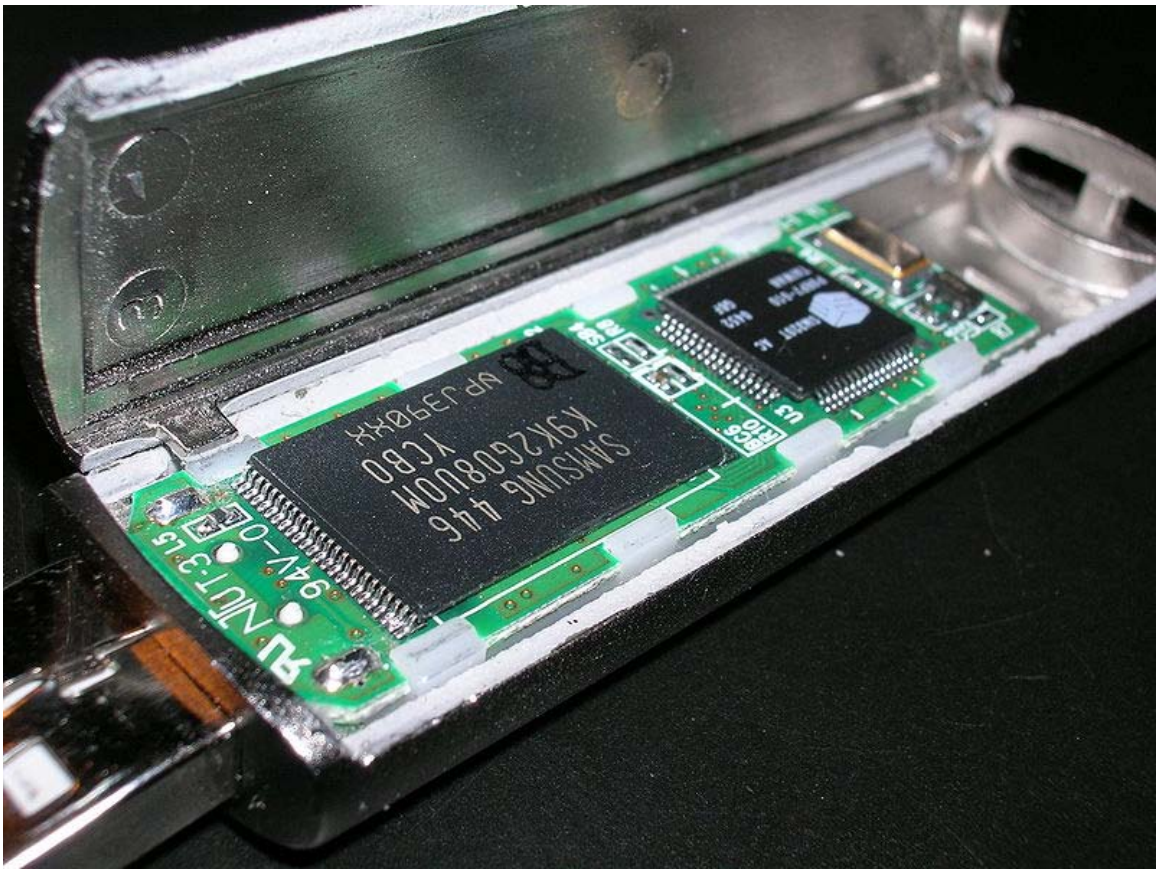
Self-modifying programs

A computer program in execution is normally treated as being different from the data the program operates on. However, in some cases this distinction is blurred when a computer program modifies itself. The modified computer program is subsequently executed as part of the same program. Self-modifying code is possible for programs written in machine code, assembly language, Lisp, C, COBOL, PL/1, Prolog and JavaScript (the eval feature) among others.

Execution and storage

Typically, computer programs are stored in non-volatile memory until requested either directly or indirectly to be executed by the computer user. Upon such a request, the program is loaded into random access memory, by a computer program called an operating system, where it can be accessed directly by the central processor. The central processor then executes ("runs") the program, instruction by instruction, until termination. A program in execution is called a process. Termination is either by normal self-termination or by error — software or hardware error.

Embedded programs



The microcontroller on the right of this USB flash drive is controlled with embedded firmware.

Some computer programs are embedded into hardware. A stored-program computer requires an initial computer program stored in its read-only memory to boot. The boot process is to identify and initialize all aspects of the system, from processor registers to device controllers to memory contents. Following the initialization process, this initial computer program loads the operating system and sets the program counter to begin normal operations. Independent of the host computer, a hardware device might have embedded firmware to control its operation. Firmware is used when the computer

program is rarely or never expected to change, or when the program must not be lost when the power is off.

Manual programming

Switches for manual input on a Data General Nova 3

Computer programs historically were manually input to the central processor via switches. An instruction was represented by a configuration of on/off settings. After setting the configuration, an execute button was pressed. This process was then repeated. Computer programs also historically were manually input via paper tape or punched cards. After the medium was loaded, the starting address was set via switches and the execute button pressed.

Automatic program generation

Generative programming is a style of computer programming that creates source code through generic classes, prototypes, templates, aspects, and code generators to improve programmer productivity. Source code is generated with programming tools such as a template processor or an integrated development environment. The simplest form of source code generator is a macro processor, such as the C preprocessor, which replaces patterns in source code according to relatively simple rules.

Software engines output source code or markup code that simultaneously become the input to another computer process. The analogy is that of one process driving another

process, with the computer code being burned as fuel. Application servers are software engines that deliver applications to client computers.

Simultaneous execution

Many operating systems support multitasking which enables many computer programs to appear to run simultaneously on one computer. Operating systems may run multiple programs through process scheduling — a software mechanism to switch the CPU among processes often so users can interact with each program while it runs. Within hardware, modern day multiprocessor computers or computers with multicore processors may run multiple programs.

One computer program can calculate simultaneously more than one operation using threads or separate processes. Multithreading processors are optimized to execute multiple threads efficiently.

Functional categories

Computer programs may be categorized along functional lines. The main functional categories are system software and application software. System software includes the operating system which couples computer hardware with application software. The purpose of the operating system is to provide an environment in which application software executes in a convenient and efficient manner. In addition to the operating system, system software includes utility programs that help manage and tune the computer. If a computer program is not system software then it is application software. Application software includes middleware, which couples the system software with the user interface. Application software also includes utility programs that help users solve application problems, like the need for sorting.

Sometimes development environments for software development are seen as a functional category on its own, especially in the context of human-computer interaction and programming language design. Development environments gather system software (such as compilers and system's batch processing scripting languages) and application software (such as IDEs) for the specific purpose of helping programmers create new programs.

Chapter 4

History of Programming Languages

Before 1940

The first programming languages predate the modern computer. At first, the languages were codes.

The Jacquard loom, invented in 1801, used holes in punched cards to represent sewing loom arm movements in order to generate decorative patterns automatically.

During a nine-month period in 1842-1843, Ada Lovelace translated the memoir of Italian mathematician Luigi Menabrea about Charles Babbage's newest proposed machine, the Analytical Engine. With the article, she appended a set of notes which specified in complete detail a method for calculating Bernoulli numbers with the Engine, recognized by some historians as the world's first computer program.

Herman Hollerith realized that he could encode information on punch cards when he observed that train conductors encode the appearance of the ticket holders on the train tickets using the position of punched holes on the tickets. Hollerith then encoded the 1890 census data on punch cards.

The first computer codes were specialized for their applications. In the first decades of the 20th century, numerical calculations were based on decimal numbers. Eventually it was realized that logic could be represented with numbers, not only with words. For example, Alonzo Church was able to express the lambda calculus in a formulaic way. The Turing machine was an abstraction of the operation of a tape-marking machine, for example, in use at the telephone companies. Turing machines set the basis for storage of programs as data in the von Neumann architecture of computers by representing a machine through a finite number. However, unlike the lambda calculus, Turing's code does not serve well as a basis for higher-level languages—its principal use is in rigorous analyses of algorithmic complexity.

Like many "firsts" in history, the first modern programming language is hard to identify. From the start, the restrictions of the hardware defined the language. Punch cards allowed 80 columns, but some of the columns had to be used for a sorting number on each card. FORTRAN included some keywords which were the same as English words, such as "IF", "GOTO" (go to) and "CONTINUE". The use of a magnetic drum for memory meant that computer programs also had to be interleaved with the rotations of the drum. Thus the programs were more hardware-dependent.

To some people, what was the first modern programming language depends on how much power and human-readability is required before the status of "programming language" is granted. Jacquard looms and Charles Babbage's Difference Engine both had simple, extremely limited languages for describing the actions that these machines should perform. One can even regard the punch holes on a player piano scroll as a limited domain-specific language, albeit not designed for human consumption.

The 1940s

In the 1940s, the first recognizably modern, electrically powered computers were created. The limited speed and memory capacity forced programmers to write hand tuned assembly language programs. It was soon discovered that programming in assembly language required a great deal of intellectual effort and was error-prone.

In 1948, Konrad Zuse published a paper about his programming language Plankalkül. However, it was not implemented in his lifetime and his original contributions were isolated from other developments.

Some important languages that were developed in this period include:

- 1943 - Plankalkül (Konrad Zuse), designed, but unimplemented for a half-century
- 1943 - ENIAC coding system, machine-specific codeset appearing in 1948
- 1949 - 1954 — a series of machine-specific mnemonic instruction sets, like ENIAC's, beginning in 1949 with C-10 for BINAC (which later evolved into UNIVAC). Each codeset, or instruction set, was tailored to a specific manufacturer.

The 1950s and 1960s

In the 1950s, the first three modern programming languages whose descendants are still in widespread use today were designed:

- FORTRAN (1955), the "**FOR**mula **TRAN**slator", invented by John Backus et al.;
- LISP [1958], the "**LIS**t **P**rocessor", invented by John McCarthy et al.;
- COBOL, the **CO**mmon **B**usiness **O**riented **L**anguage, created by the Short Range Committee, heavily influenced by Grace Hopper.

Another milestone in the late 1950s was the publication, by a committee of American and European computer scientists, of "a new language for algorithms"; the *ALGOL 60 Report* (the "**ALGO**rithmic **L**anguage"). This report consolidated many ideas circulating at the time and featured two key language innovations:

- nested block structure: code sequences and associated declarations could be grouped into blocks without having to be turned into separate, explicitly named procedures;
- lexical scoping: a block could have its own private variables, procedures and functions, invisible to code outside that block, i.e. information hiding.

Another innovation, related to this, was in how the language was described:

- a mathematically exact notation, Backus-Naur Form (BNF), was used to describe the language's syntax. Nearly all subsequent programming languages have used a variant of BNF to describe the context-free portion of their syntax.

Algol 60 was particularly influential in the design of later languages, some of which soon became more popular. The Burroughs large systems were designed to be programmed in an extended subset of Algol.

Algol's key ideas were continued, producing ALGOL 68:

- syntax and semantics became even more orthogonal, with anonymous routines, a recursive typing system with higher-order functions, etc.;
- not only the context-free part, but the full language syntax and semantics were defined formally, in terms of Van Wijngaarden grammar, a formalism designed specifically for this purpose.

Algol 68's many little-used language features (e.g. concurrent and parallel blocks) and its complex system of syntactic shortcuts and automatic type coercions made it unpopular with implementers and gained it a reputation of being *difficult*. Niklaus Wirth actually walked out of the design committee to create the simpler Pascal language.

Some important languages that were developed in this period include:

- 1951 - Regional Assembly Language
- 1952 - Autocode
- 1954 - FORTRAN
- 1954 - IPL (forerunner to LISP)
- 1955 - FLOW-MATIC (forerunner to COBOL)
- 1957 - COMTRAN (forerunner to COBOL)
- 1958 - LISP
- 1958 - ALGOL 58
- 1959 - FACT (forerunner to COBOL)
- 1959 - COBOL

- 1962 - APL
- 1962 - Simula
- 1962 - SNOBOL
- 1963 - CPL (forerunner to C)
- 1964 - BASIC
- 1964 - PL/I
- 1967 - BCPL (forerunner to C)

1967-1978: establishing fundamental paradigms

The period from the late 1960s to the late 1970s brought a major flowering of programming languages. Most of the major language paradigms now in use were invented in this period:

- **Simula**, invented in the late 1960s by Nygaard and Dahl as a superset of Algol 60, was the first language designed to support object-oriented programming.
- **C**, an early systems programming language, was developed by Dennis Ritchie and Ken Thompson at Bell Labs between 1969 and 1973.
- **Smalltalk** (mid 1970s) provided a complete ground-up design of an object-oriented language.
- **Prolog**, designed in 1972 by Colmerauer, Roussel, and Kowalski, was the first logic programming language.
- **ML** built a polymorphic type system (invented by Robin Milner in 1973) on top of Lisp, pioneering statically typed functional programming languages.

Each of these languages spawned an entire family of descendants, and most modern languages count at least one of them in their ancestry.

The 1960s and 1970s also saw considerable debate over the merits of "structured programming", which essentially meant programming without the use of Goto. This debate was closely related to language design: some languages did not include GOTO, which forced structured programming on the programmer. Although the debate raged hotly at the time, nearly all programmers now agree that, even in languages that provide GOTO, it is bad programming style to use it except in rare circumstances. As a result, later generations of language designers have found the structured programming debate tedious and even bewildering.

Some important languages that were developed in this period include:

- 1968 - Logo
- 1970 - Pascal
- 1970 - Forth
- 1972 - C
- 1972 - Smalltalk
- 1972 - Prolog
- 1973 - ML

- 1975 - Scheme
- 1978 - SQL (initially only a query language, later extended with programming constructs)

The 1980s: consolidation, modules, performance

The 1980s were years of relative consolidation in imperative languages. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called fifth-generation programming languages that incorporated logic programming constructs. The functional languages community moved to standardize ML and Lisp. Research in Miranda, a functional language with lazy evaluation, began to take hold in this decade.

One important new trend in language design was an increased focus on programming for large-scale systems through the use of *modules*, or large-scale organizational units of code. Modula, Ada, and ML all developed notable module systems in the 1980s. Module systems were often wedded to generic programming constructs---generics being, in essence, parameterized modules.

Although major new paradigms for imperative programming languages did not appear, many researchers expanded on the ideas of prior languages and adapted them to new contexts. For example, the languages of the Argus and Emerald systems adapted object-oriented programming to distributed systems.

The 1980s also brought advances in programming language implementation. The RISC movement in computer architecture postulated that hardware should be designed for compilers rather than for human assembly programmers. Aided by processor speed improvements that enabled increasingly aggressive compilation techniques, the RISC movement sparked greater interest in compilation technology for high-level languages.

Language technology continued along these lines well into the 1990s.

Some important languages that were developed in this period include:

- 1980 - C++ (as C with classes, name changed in July 1983)
- 1983 - Objective-C
- 1983 - Ada
- 1984 - Common Lisp
- 1985 - Eiffel
- 1986 - Erlang
- 1987 - Perl
- 1988 - Tcl
- 1989 - FL (Backus)

The 1990s: the Internet age

The 1990s saw no fundamental novelty in imperative languages, but much recombination and maturation of old ideas. This era began the spread of functional languages. A big driving philosophy was programmer productivity. Many "rapid application development" (RAD) languages emerged, which usually came with an IDE, garbage collection, and were descendants of older languages. All such languages were object-oriented. These included Object Pascal, Visual Basic, and Java. Java in particular received much attention. More radical and innovative than the RAD languages were the new scripting languages. These did not directly descend from other languages and featured new syntaxes and more liberal incorporation of features. Many consider these scripting languages to be more productive than even the RAD languages, but often because of choices that make small programs simpler but large programs more difficult to write and maintain. Nevertheless, scripting languages came to be the most prominent ones used in connection with the Web.

Some important languages that were developed in this period include:

- 1990 - Haskell
- 1991 - Python
- 1991 - Visual Basic
- 1993 - Ruby
- 1993 - Lua
- 1994 - CLOS (part of ANSI Common Lisp)
- 1995 - Java
- 1995 - Delphi (Object Pascal)
- 1995 - JavaScript
- 1995 - PHP
- 1997 - Rebol
- 1999 - D

Current trends

Programming language evolution continues, in both industry and research. Some of the current trends include:

- Mechanisms for adding security and reliability verification to the language: extended static checking, information flow control, static thread safety.
- Alternative mechanisms for modularity: mixins, delegates, aspects.
- Component-oriented software development.
- Constructs to support concurrent and distributed programming.
- Metaprogramming, reflection or access to the abstract syntax tree
- Increased emphasis on distribution and mobility.
- Integration with databases, including XML and relational databases.

- Support for Unicode so that source code (program text) is not restricted to those characters contained in the ASCII character set; allowing, for example, use of non-Latin-based scripts or extended punctuation.
- XML for graphical interface (XUL, XAML).

Some important languages developed during this period include:

- 2001 - C#
- 2001 - Visual Basic .NET
- 2002 - F#
- 2003 - Scala
- 2003 - Factor
- 2006 - Windows Power Shell
- 2007 - Clojure
- 2007 - Groovy
- 2009 - Go

Prominent people in the history of programming languages

- John Backus, inventor of Fortran.
- Alan Cooper, developer of Visual Basic.
- Edsger W. Dijkstra, developed the framework for structured programming.
- James Gosling, developer of Oak, the precursor of Java.
- Anders Hejlsberg, developer of Turbo Pascal, Delphi and C#.
- Grace Hopper, developer of Flow-Matic, influencing COBOL.
- Kenneth E. Iverson, developer of APL, and co-developer of J along with Roger Hui.
- Bill Joy, inventor of vi, early author of BSD Unix, and originator of SunOS, which became Solaris.
- Alan Kay, pioneering work on object-oriented programming, and originator of Smalltalk.
- Brian Kernighan, co-author of the first book on the C programming language with Dennis Ritchie, coauthor of the AWK and AMPL programming languages.
- John McCarthy, inventor of LISP.
- John von Neumann, originator of the operating system concept.
- Dennis Ritchie, inventor of C (programming language). Unix Operating System , Plan 9 Operating System.
- Bjarne Stroustrup, developer of C++.
- Ken Thompson, inventor of B , Go Programming Language , Inferno Programming Language.
- Niklaus Wirth, inventor of Pascal, Modula and Oberon.
- Larry Wall, creator of Perl and Perl 6
- Guido van Rossum, creator of Python
- Yukihiro Matsumoto, creator of Ruby

Chapter 5

Abstraction (Computer Science)

In computer science, **abstraction** is the process by which data and programs are defined with a representation similar to its meaning (semantics), while hiding away the implementation details. Abstraction tries to reduce and factor out details so that the programmer can focus on a few concepts at a time. A system can have several *abstraction layers* whereby different meanings and amounts of detail are exposed to the programmer. For example, low-level abstraction layers expose details of the hardware where the program is run, while high-level layers deal with the business logic of the program.

The following English definition of abstraction helps to understand how this term applies to computer science, IT and objects:

abstraction - a concept or idea not associated with any specific instance

The concept originated by analogy with abstraction in mathematics. The mathematical technique of abstraction begins with mathematical definitions; this has the fortunate effect of finessing some of the vexing philosophical issues of abstraction. For example, in both computing and in mathematics, numbers are concepts in the programming languages, as founded in mathematics. Implementation details depend on the hardware and software, but this is not a restriction because the computing concept of number is still based on the mathematical concept.

In computer programming, abstraction can apply to control or to data: **Control abstraction** is the abstraction of actions while **data abstraction** is that of data structures.

- Control abstraction involves the use of subprograms and related concepts control flows
- Data abstraction allows handling data bits in meaningful ways. For example, it is the basic motivation behind datatype.

One can regard the notion of an object (from object-oriented programming) as an attempt to combine abstractions of data and code.

The same abstract definition can be used as a common interface for a family of objects with different implementations and behaviors but which share the same meaning. The inheritance mechanism in object-oriented programming can be used to define an abstract class as the common interface.

The recommendation that programmers use abstractions whenever suitable in order to avoid duplication (usually of code) is known as the abstraction principle. The requirement that a programming language provide suitable abstractions is also called the abstraction principle.

Rationale

Computing mostly operates independently of the concrete world: The hardware implements a model of computation that is interchangeable with others. The software is structured in architectures to enable humans to create the enormous systems by concentration on a few issues at a time. These architectures are made of specific choices of abstractions. Greenspun's Tenth Rule is an aphorism on how such an architecture is both inevitable and complex.

A central form of abstraction in computing is language abstraction: new artificial languages are developed to express specific aspects of a system. *Modeling languages* help in planning. *Computer languages* can be processed with a computer. An example of this abstraction process is the generational development of programming languages from the machine language to the assembly language and the high-level language. Each stage can be used as a stepping stone for the next stage. The language abstraction continues for example in scripting languages and domain-specific programming languages.

Within a programming language, some features let the programmer create new abstractions. These include the subroutine, the module, and the software component. Some other abstractions such as software design patterns and architectural styles remain invisible to a programming language and operate only in the design of a system.

Some abstractions try to limit the breadth of concepts a programmer needs by completely hiding the abstractions they in turn are built on. Joel Spolsky has criticised these efforts by claiming that all abstractions are *leaky* — that they can never completely hide the details below; however this does not negate the usefulness of abstraction. Some abstractions are designed to interoperate with others, for example a programming language may contain a foreign function interface for making calls to the lower-level language.

Language features

Programming languages

Different programming languages provide different types of abstraction, depending on the intended applications for the language. For example:

- In object-oriented programming languages such as C++, Object Pascal, or Java, the concept of **abstraction** has itself become a declarative statement - using the keywords `virtual` (in C++) or `abstract` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.
- Functional programming languages commonly exhibit abstractions related to functions, such as lambda abstractions (making a term into a function of some variable), higher-order functions (parameters are functions), bracket abstraction (making a term into a function of a variable).

Specification methods

Analysts have developed various methods to formally specify software systems. Some known methods include:

- Abstract-model based method (VDM, Z);
- Algebraic techniques (Larch, CLEAR, OBJ, ACT ONE, CASL);
- Process-based techniques (LOTOS, SDL, Estelle);
- Trace-based techniques (SPECIAL, TAM);
- Knowledge-based techniques (Refine, Gist).

Specification languages

Specification languages generally rely on abstractions of one kind or another, since specifications are typically defined earlier in a project (and at a more abstract level) than an eventual implementation. The UML specification language, for example, allows the definition of *abstract* classes, which remain abstract during the architecture and specification phase of the project.

Control abstraction

Programming languages offer control abstraction as one of the main purposes of their use. Computer machines understand operations at the very low level such as moving some bits from one location of the memory to another location and producing the sum of two sequences of bits. Programming languages allow this to be done in the higher level. For example, consider this statement written in a Pascal-like fashion:

```
a := (1 + 2) * 5
```

To a human, this seems a fairly simple and obvious calculation (*"one plus two is three, times five is fifteen"*). However, the low-level steps necessary to carry out this evaluation, and return the value "15", and then assign that value to the variable "a", are actually quite subtle and complex. The values need to be converted to binary representation (often a much more complicated task than one would think) and the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of one register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of "15" to the variable labeled "a", so that "a" can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label and the resultant location in physical or virtual memory, storing the binary representation of "15" to that memory location, etc.

Without control abstraction, a programmer would need to specify *all* the register/binary-level steps each time he simply wanted to add or multiply a couple of numbers and assign the result to a variable. Such duplication of effort has two serious negative consequences:

1. it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed
2. it forces the programmer to program for the particular hardware and instruction set

Structured programming

Structured programming involves the splitting of complex program tasks into smaller pieces with clear flow-control and interfaces between components, with reduction of the complexity potential for side-effects.

In a simple program, this may aim to ensure that loops have single or obvious exit points and (where possible) to have single exit points from functions and procedures.

In a larger system, it may involve breaking down complex tasks into many different modules. Consider a system which handles payroll on ships and at shore offices:

- The uppermost level may feature a menu of typical end-user operations.
- Within that could be standalone executables or libraries for tasks such as signing on and off employees or printing checks.
- Within each of those standalone components there could be many different source files, each containing the program code to handle a part of the problem, with only selected interfaces available to other parts of the program. A sign on program could have source files for each data entry screen and the database interface (which may itself be a standalone third party library or a statically linked set of library routines).

- Either the database or the payroll application also has to initiate the process of exchanging data with between ship and shore, and that data transfer task will often contain many other components.

These layers produce the effect of isolating the implementation details of one component and its assorted internal methods from the others. Object-oriented programming embraced and extended this concept.

Data abstraction

Data abstraction enforces a clear separation between the *abstract* properties of a data type and the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type—the *interface* to the data type—while the concrete implementation is kept entirely private, and indeed can change, for example to incorporate efficiency improvements over time. The idea is that such changes are not supposed to have any impact on client code, since they involve no difference in the abstract behaviour.

For example, one could define an abstract data type called *lookup table* which uniquely associates *keys* with *values*, and in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of (key:value) pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the first place, since any changes there can have major impacts on client code. As one way to look at this: the interface forms a *contract* on agreed behaviour between the data type and client code; anything not spelled out in the contract is subject to change without notice.

Languages that implement data abstraction include Ada and Modula-2. Object-oriented languages are commonly claimed to offer data abstraction; however, their inheritance concept tends to put information in the interface that more properly belongs in the implementation; thus, changes to such information ends up impacting client code, leading directly to the Fragile binary interface problem.

Abstraction in object oriented programming

In object-oriented programming theory, **abstraction** involves the facility to define objects that represent abstract "actors" that can perform work, report on and change their state, and "communicate" with other objects in the system. The term encapsulation refers to the hiding of state details, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, and standardizing the way that different data types interact, is the beginning of **abstraction**. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called polymorphism. When it proceeds in the opposite direction,

inside the types or classes, structuring them to simplify a complex set of relationships, it is called delegation or inheritance.

Various object-oriented programming languages offer similar facilities for abstraction, all to support a general strategy of polymorphism in object-oriented programming, which includes the substitution of one type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or loadtime. This would leave only a minimum of such bindings to change at run-time.

Common Lisp Object System or self, for example, feature less of a class-instance distinction and more use of delegation for polymorphism. Individual objects and functions are abstracted more flexibly to better fit with a shared functional heritage from Lisp.

C++ exemplifies another extreme: it relies heavily on templates and overloading and other static bindings at compile-time, which in turn has certain flexibility problems.

Although these examples offer alternate strategies for achieving the same abstraction, they do not fundamentally alter the need to support abstract nouns in code - all programming relies on an ability to abstract verbs as functions, nouns as data structures, and either as processes.

Consider for example a sample Java fragment to represent some common farm "animals" to a level of abstraction suitable to model simple aspects of their hunger and feeding. It defines an `Animal` class to represent both the state of the animal and its functions:

```
public class Animal extends LivingThing
{
     private Location loc;
     private double energyReserves;

     private boolean isHungry() {
         return energyReserves < 2.5;
     }
     private void eat(Food f) {
         // Consume food
         energyReserves += f.getCalories();
     }
     private void moveTo(Location l) {
         // Move to new location
         loc = l;
     }
}
```

With the above definition, one could create objects of type `Animal` and call their methods like this:

```
thePig = new Animal();
theCow = new Animal();
```

```
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

In the above example, the class *Animal* is an abstraction used in place of an actual animal, *LivingThing* is a further abstraction (in this case a generalisation) of *Animal*.

If one requires a more differentiated hierarchy of animals — to differentiate, say, those who provide milk from those who provide nothing except meat at the end of their lives — that is an intermediary level of abstraction, probably DairyAnimal (cows, goats) who would eat foods suitable to giving good milk, and Animal (pigs, steers) who would eat foods to give the best meat-quality.

Such an abstraction could remove the need for the application coder to specify the type of food, so s/he could concentrate instead on the feeding schedule. The two classes could be related using inheritance or stand alone, and the programmer could define varying degrees of polymorphism between the two types. These facilities tend to vary drastically between languages, but in general each can achieve anything that is possible with any of the others. A great many operation overloads, data type by data type, can have the same effect at compile-time as any degree of inheritance or other means to achieve polymorphism. The class notation is simply a coder's convenience.

Object-oriented design

Decisions regarding what to abstract and what to keep under the control of the coder become the major concern of object-oriented design and domain analysis—actually determining the relevant relationships in the real world is the concern of object-oriented analysis or legacy analysis.

In general, to determine appropriate abstraction, one must make many small decisions about scope (domain analysis), determine what other systems one must cooperate with (legacy analysis), then perform a detailed object-oriented analysis which is expressed within project time and budget constraints as an object-oriented design. In our simple example, the domain is the barnyard, the live pigs and cows and their eating habits are the legacy constraints, the detailed analysis is that coders must have the flexibility to feed the animals what is available and thus there is no reason to code the type of food into the class itself, and the design is a single simple Animal class of which pigs and cows are instances with the same functions. A decision to differentiate DairyAnimal would change the detailed analysis but the domain and legacy analysis would be unchanged—thus it is entirely under the control of the programmer, and we refer to abstraction in object-oriented programming as distinct from abstraction in domain or legacy analysis.

Considerations

When discussing formal semantics of programming languages, formal methods or abstract interpretation, **abstraction** refers to the act of considering a less detailed, but safe, definition of the observed program behaviors. For instance, one may observe only the final result of program executions instead of considering all the intermediate steps of executions. Abstraction is defined to a **concrete** (more precise) model of execution.

Abstraction may be **exact** or **faithful** with respect to a property if one can answer a question about the property equally well on the concrete or abstract model. For instance, if we wish to know what the result of the evaluation of a mathematical expression involving only integers +, -, ×, is worth modulo *n*, we need only perform all operations modulo *n* (a familiar form of this abstraction is casting out nines).

Abstractions, however, though not necessarily **exact**, should be **sound**. That is, it should be possible to get sound answers from them—even though the abstraction may simply yield a result of undecidability. For instance, we may abstract the students in a class by their minimal and maximal ages; if one asks whether a certain person belongs to that class, one may simply compare that person's age with the minimal and maximal ages; if his age lies outside the range, one may safely answer that the person does not belong to the class; if it does not, one may only answer "I don't know".

The level of abstraction included in a programming language can influence its overall usability. The Cognitive dimensions framework includes the concept of *abstraction gradient* in a formalism. This framework allows the designer of a programming language to study the trade-offs between abstraction and other characteristics of the design, and how changes in abstraction influence the language usability.

Abstractions can prove useful when dealing with computer programs, because non-trivial properties of computer programs are essentially undecidable. As a consequence, automatic methods for deriving information on the behavior of computer programs either have to drop termination (on some occasions, they may fail, crash or never yield out a result), soundness (they may provide false information), or precision (they may answer "I don't know" to some questions).

Abstraction is the core concept of abstract interpretation. Model checking generally takes place on abstract versions of the studied systems.

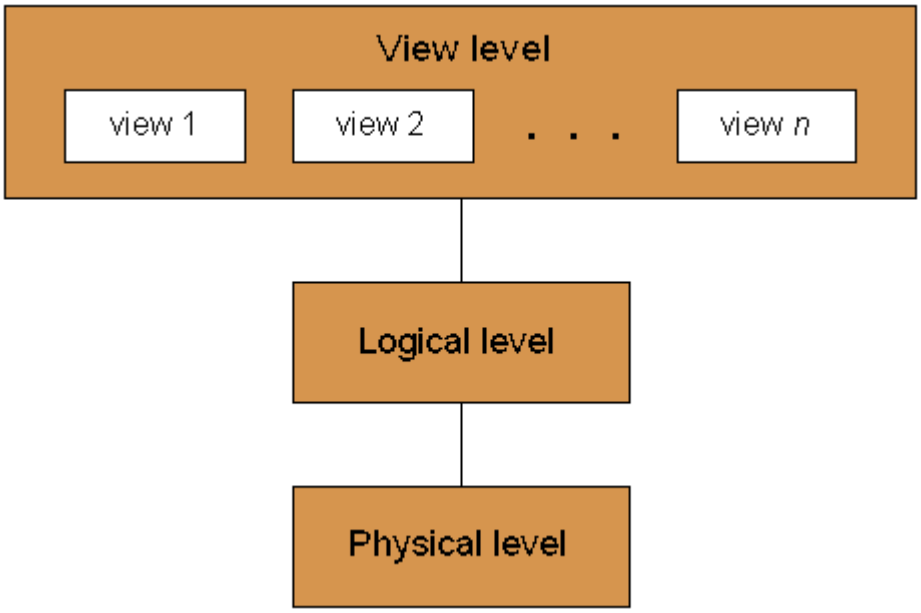
Levels of abstraction

Computer science commonly presents *levels* (or, less commonly, *layers*) of abstraction, wherein each level represents a different model of the same information and processes, but uses a system of expression involving a unique set of objects and compositions that apply only to a particular domain. Each relatively abstract, "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. For example, gates build on electronic circuits, binary on gates, machine

language on binary, programming language on machine language, applications and operating systems on programming languages. Each level is embodied, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Database systems

Since many users of database systems lack in-depth familiarity with computer data-structures, database developers often hide complexity through the following levels:



Data abstraction levels of a database system

Physical level: The lowest level of abstraction describes *how* a system actually stores data. The physical level describes complex low-level data structures in detail.

Logical level: The next higher level of abstraction describes *what* data the database stores, and what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. This referred to as Physical Data Independence. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.

View level: The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide many views for the same database.

Layered architecture

The ability to provide a design of different levels of abstraction can

- simplify the design considerably
- enable different role players to effectively work at various levels of abstraction

Systems design and business process design can both use this. Some design processes specifically generate designs that contain various levels of abstraction.

Layered architecture partitions the concerns of the application into stacked groups (layers). it is a technique used in designing computer software, hardware, and communications in which system or network components are isolated in layers so that changes can be made in one layer without affecting the others.

Chapter 6

Assembly Language

An **assembly language** is a low-level programming language for computers, microprocessors, microcontrollers, and other programmable devices. It implements a symbolic representation of the machine codes and other constants needed to program a given CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on mnemonics that symbolize processing steps (instructions), processor registers, memory locations, and other language features. An assembly language is thus specific to a certain physical (or virtual) computer architecture. This is in contrast to most high-level programming languages, which, ideally, are portable.

A utility program called an *assembler* is used to translate assembly language statements into the target computer's machine code. The assembler performs a more or less isomorphic translation (a one-to-one mapping) from mnemonic statements into machine instructions and data. This is in contrast with high-level languages, in which a single statement generally results in many machine instructions.

Many sophisticated assemblers offer additional mechanisms to facilitate program development, control the assembly process, and aid debugging. In particular, most modern assemblers include a macro facility (described below), and are called *macro assemblers*.

Key concepts

Assembler

Typically a modern **assembler** creates object code by translating assembly instruction mnemonics into opcodes, and by resolving symbolic names for memory locations and other entities. The use of symbolic references is a key feature of assemblers, saving tedious calculations and manual address updates after program modifications. Most

assemblers also include macro facilities for performing textual substitution—e.g., to generate common short sequences of instructions as inline, instead of *called* subroutines.

Assemblers are generally simpler to write than compilers for high-level languages, and have been available since the 1950s. Modern assemblers, especially for RISC architectures, such as SPARC or POWER, as well as x86 and x86-64, optimize Instruction scheduling to exploit the CPU pipeline efficiently.

Number of passes

There are two types of assemblers based on how many passes through the source are needed to produce the executable program.

- One-pass assemblers go through the source code once and assume that all symbols will be defined before any instruction that references them.
- Two-pass assemblers create a table with all symbols and their values in the first pass, then use the table in a second pass to generate code. The assembler must at least be able to determine the length of each instruction on the first pass so that the addresses of symbols can be calculated.

The advantage of a one-pass assembler is speed, which is not as important as it once was with advances in computer speed and abilities. The advantage of the two-pass assembler is that symbols can be defined anywhere in program source code. This lets programs be defined in more logical and meaningful ways, making two-pass assembler programs easier to read and maintain.

High-level assemblers

More sophisticated high-level assemblers provide language abstractions such as:

- Advanced control structures
- High-level procedure/function declarations and invocations
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing (although available on ordinary assemblers since late 1950s for IBM 700 series and since 1960's for IBM/360, amongst other machines)
- Object-oriented programming features such as classes, objects, abstraction, polymorphism, and inheritance

Use of the term

Note that, in normal professional usage, the term *assembler* is often used ambiguously: It can refer to an assembly language itself, as well as to an assembler utility. Thus: "CP/CMS was written in S/360 assembler" as well as "ASM-H was a widely-used S/370 assembler."

Assembly language

A program written in assembly language consists of a series of mnemonic statements and meta-statements (known variously as directives, pseudo-instructions and pseudo-ops), comments and data. These are translated by an assembler to a stream of executable instructions that can be loaded into memory and executed. Assemblers can also be used to produce blocks of data, from formatted and commented source code, to be used by other code.

Take for example, the instruction that tells an x86/IA-32 processor to move an immediate 8-bit value into a register. The binary code for this instruction is 10110 followed by a 3-bit identifier for which register to use. The identifier for the *AL* register is 000, so the following machine code loads the *AL* register with the data 01100001.

```
10110000 01100001
```

This binary computer code can be made more human-readable by expressing it in hexadecimal as follows

```
B0 61
```

Here, `B0` means 'Move a copy of the following value into *AL'*, and `61` is a hexadecimal representation of the value 01100001, which is 97 in decimal. Intel assembly language provides the mnemonic MOV (an abbreviation of *move*) for instructions such as this, so the machine code above can be written as follows in assembly language, complete with an explanatory comment if required, after the semicolon. This is much easier to read and to remember.

```
MOV AL, 61h       ; Load AL with 97 decimal (61 hex)
```

At one time, many assembly language mnemonics were three letter abbreviations, such as JMP for *jump*, INC for *increment*, etc. Modern processors have a much larger instruction set and many mnemonics are now longer, for example FPATAN for "*floating point partial arctangent*" and BOUND for "*check array index against bounds*". Many assembly language statements consist of an opcode mnemonic followed by a comma-separated list of data, arguments or parameters.

The same mnemonic MOV refers to a family of related opcodes to do with loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers. The opcode 10110000 (`B0`) copies an 8-bit value into the *AL* register, while 10110001 (`B1`) moves it into *CL* and 10110010 (`B2`) does so into *DL*. Assembly language examples for these follow.

```
MOV AL, 1h        ; Load AL with immediate value 1
MOV CL, 2h        ; Load CL with immediate value 2
MOV DL, 3h        ; Load DL with immediate value 3
```

The syntax of MOV can also be more complex as the following examples show.

```
MOV EAX, [EBX]        ; Move the 4 bytes in memory at the address
contained in EBX into EAX
MOV [ESI+EAX], CL ; Move the contents of CL into the byte at address
ESI+EAX
```

In each case, the MOV mnemonic is translated directly into an opcode in the ranges 88-8E, A0-A3, B0-B8, C6 or C7 by an assembler, and the programmer does not have to know or remember which.

Transforming assembly language into machine code is the job of an assembler, and the reverse can at least partially be achieved by a disassembler. Unlike high-level languages, there is usually a one-to-one correspondence between simple assembly statements and machine language instructions. However, in some cases, an assembler may provide *pseudoinstructions* (essentially macros) which expand into several machine language instructions to provide commonly needed functionality. For example, for a machine that lacks a "branch if greater or equal" instruction, an assembler may provide a pseudoinstruction that expands to the machine's "set if less than" and "branch if zero (on the result of the set instruction)". Most full-featured assemblers also provide a rich macro language (discussed below) which is used by vendors and programmers to generate more complex code and data sequences.

Each computer architecture and processor architecture usually has its own machine language. On this level, each instruction is simple enough to be executed using a relatively small number of electronic circuits. Computers differ by the number and type of operations they support. For example, a new 64-bit machine would have different circuitry from a 32-bit machine. They may also have different sizes and numbers of registers, and different representations of data types in storage. While most general-purpose computers are able to carry out essentially the same functionality, the ways they do so differ; the corresponding assembly languages reflect these differences.

Multiple sets of mnemonics or assembly-language syntax may exist for a single instruction set, typically instantiated in different assembler programs. In these cases, the most popular one is usually that supplied by the manufacturer and used in its documentation.

Language design

Basic elements

There is a large degree of diversity in the way the authors of assemblers categorize statements and in the nomenclature that they use. In particular, some describe anything other than a machine mnemonic or extended mnemonic as a pseudo-operation (pseudo-op). A typical assembly language consists of 3 types of instruction statements that are used to define program operations:

- Opcode mnemonics
- Data sections
- Assembly directives

Opcode mnemonics and extended mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level language. Generally, a mnemonic is a symbolic name for a single executable machine language instruction (an opcode), and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an *operation* or *opcode* plus zero or more *operands*. Most instructions refer to a single value, or a pair of values. Operands can be immediate (typically one byte values, coded in the instruction itself), registers specified in the instruction, implied or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works. *Extended mnemonics* are often used to specify a combination of an opcode with a specific operand, e.g., the System/360 assemblers use **B** as an extended mnemonic for **BC** with a mask of 15 and **NOP** for **BC** with a mask of 0.

Extended mnemonics are often used to support specialized uses of instructions, often for purposes not obvious from the instruction name. For example, many CPU's do not have an explicit NOP instruction, but do have instructions that can be used for the purpose. In 8086 CPUs the instruction *xchg ax,ax* is used for *nop*, with *nop* being a pseudo-opcode to encode the instruction *xchg ax,ax*. Some disassemblers recognize this and will decode the *xchg ax,ax* instruction as *nop*. Similarly, IBM assemblers for System/360 and System/360 use the extended mnemonics *NOP* and *NOPR* for *BC* and *BCR* with zero masks.

Some assemblers also support simple built-in macro-instructions that generate two or more machine instructions. For instance, with some Z80 assemblers the instruction *ld hl,bc* is recognized to generate *ld l,c* followed by *ld h,b*. These are sometimes known as *pseudo-opcodes*.

Data sections

There are instructions used to define data elements to hold data and variables. They define the type of data, the length and the alignment of data. These instructions can also define whether the data is available to outside programs (programs assembled separately) or only to the program in which the data section is defined. Some assemblers classify these as pseudo-ops.

Assembly directives

Assembly directives, also called pseudo opcodes, pseudo-operations or pseudo-ops, are instructions that are executed by an assembler at assembly time, not by a CPU at run time. They can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled different ways, perhaps for different

applications. They also can be used to manipulate presentation of a program to make it easier to read and maintain.

(For example, directives would be used to reserve storage areas and optionally their initial contents.) The names of directives often start with a dot to distinguish them from machine instructions.

Symbolic assemblers let programmers associate arbitrary names (*labels* or *symbols*) with memory locations. Usually, every constant and variable is given a name so instructions can reference those locations by name, thus promoting self-documenting code. In executable code, the name of each subroutine is associated with its entry point, so any calls to a subroutine can use its name. Inside subroutines, GOTO destinations are given labels. Some assemblers support *local symbols* which are lexically distinct from normal symbols (e.g., the use of "10$" as a GOTO destination).

Some assemblers provide flexible symbol management, letting programmers manage different namespaces, automatically calculate offsets within data structures, and assign labels that refer to literal values or the result of simple computations performed by the assembler. Labels can also be used to initialize constants and variables with relocatable addresses.

Assembly languages, like most other computer languages, allow comments to be added to assembly source code that are ignored by the assembler. Good use of comments is even more important with assembly code than with higher-level languages, as the meaning and purpose of a sequence of instructions is harder to decipher from the code itself.

Wise use of these facilities can greatly simplify the problems of coding and maintaining low-level code. *Raw* assembly source code as generated by compilers or disassemblers—code without any comments, meaningful symbols, or data definitions—is quite difficult to read when changes must be made.

Macros

Many assemblers support *predefined macros*, and others support *programmer-defined* (and repeatedly re-definable) macros involving sequences of text lines in which variables and constants are embedded. This sequence of text lines may include opcodes or directives. Once a macro has been defined its name may be used in place of a mnemonic. When the assembler processes such a statement, it replaces the statement with the text lines associated with that macro, then processes them as if they existed in the source code file (including, in some assemblers, expansion of any macros existing in the replacement text).

Since macros can have 'short' names but expand to several or indeed many lines of code, they can be used to make assembly language programs appear to be far shorter, requiring fewer lines of source code, as with higher level languages. They can also be used to add

higher levels of structure to assembly programs, optionally introduce embedded debugging code via parameters and other similar features.

Many assemblers have built-in (or *predefined*) macros for system calls and other special code sequences, such as the generation and storage of data realized through advanced bitwise and boolean operations used in gaming, software security, data management, and cryptography.

Macro assemblers often allow macros to take parameters. Some assemblers include quite sophisticated macro languages, incorporating such high-level language elements as optional parameters, symbolic variables, conditionals, string manipulation, and arithmetic operations, all usable during the execution of a given macro, and allowing macros to save context or exchange information. Thus a macro might generate a large number of assembly language instructions or data definitions, based on the macro arguments. This could be used to generate record-style data structures or "unrolled" loops, for example, or could generate entire algorithms based on complex parameters. An organization using assembly language that has been heavily extended using such a macro suite can be considered to be working in a higher-level language, since such programmers are not working with a computer's lowest-level conceptual elements.

Macros were used to customize large scale software systems for specific customers in the mainframe era and were also used by customer personnel to satisfy their employers' needs by making specific versions of manufacturer operating systems. This was done, for example, by systems programmers working with IBM's Conversational Monitor System / Virtual Machine (CMS/VM) and with IBM's "real time transaction processing" add-ons, CICS, Customer Information Control System, and ACP/TPF, the airline/financial system that began in the 1970s and still runs many large computer reservations systems (CRS) and credit card systems today.

It was also possible to use solely the macro processing abilities of an assembler to generate code written in completely different languages, for example, to generate a version of a program in COBOL using a pure macro assembler program containing lines of COBOL code inside assembly time operators instructing the assembler to generate arbitrary code.

This was because, as was realized in the 1960s, the concept of "macro processing" is independent of the concept of "assembly", the former being in modern terms more word processing, text processing, than generating object code. The concept of macro processing appeared, and appears, in the C programming language, which supports "preprocessor instructions" to set variables, and make conditional tests on their values. Note that unlike certain previous macro processors inside assemblers, the C preprocessor was not Turing-complete because it lacked the ability to either loop or "go to", the latter allowing programs to loop.

Despite the power of macro processing, it fell into disuse in many high level languages (a major exception being C/C++) while remaining a perennial for assemblers. This was

because many programmers were rather confused by macro parameter substitution and did not disambiguate macro processing from assembly and execution.

Macro parameter substitution is strictly by name: at macro processing time, the value of a parameter is textually substituted for its name. The most famous class of bugs resulting was the use of a parameter that itself was an expression and not a simple name when the macro writer expected a name. In the macro: `foo: macro a load a*b` the intention was that the caller would provide the name of a variable, and the "global" variable or constant b would be used to multiply "a". If foo is called with the parameter `a-c`, the macro expansion of `load a-c*b` occurs. To avoid any possible ambiguity, users of macro processors can parenthesize formal parameters inside macro definitions, or callers can parenthesize the input parameters.

PL/I and C/C++ feature macros, but this facility can only manipulate text. On the other hand, homoiconic languages, such as Lisp, Prolog, and Forth, retain the power of assembly language macros because they are able to manipulate their own code as data.

Support for structured programming

Some assemblers have incorporated structured programming elements to encode execution flow. The earliest example of this approach was in the Concept-14 macro set, originally proposed by Dr. H.D. Mills (March, 1970), and implemented by Marvin Kessler at IBM's Federal Systems Division, which extended the S/360 macro assembler with IF/ELSE/ENDIF and similar control flow blocks. This was a way to reduce or eliminate the use of GOTO operations in assembly code, one of the main factors causing spaghetti code in assembly language. This approach was widely accepted in the early 80s (the latter days of large-scale assembly language use).

A curious design was A-natural, a "stream-oriented" assembler for 8080/Z80 processors from Whitesmiths Ltd. (developers of the Unix-like Idris operating system, and what was reported to be the first commercial C compiler). The language was classified as an assembler, because it worked with raw machine elements such as opcodes, registers, and memory references; but it incorporated an expression syntax to indicate execution order. Parentheses and other special symbols, along with block-oriented structured programming constructs, controlled the sequence of the generated instructions. A-natural was built as the object language of a C compiler, rather than for hand-coding, but its logical syntax won some fans.

There has been little apparent demand for more sophisticated assemblers since the decline of large-scale assembly language development. In spite of that, they are still being developed and applied in cases where resource constraints or peculiarities in the target system's architecture prevent the effective use of higher-level languages.

Use of assembly language

Historical perspective

Assembly languages were first developed in the 1950s, when they were referred to as second generation programming languages. For example, SOAP (Symbolic Optimal Assembly Program) was a 1957 assembly language for the IBM 650 computer. Assembly languages eliminated much of the error-prone and time-consuming first-generation programming needed with the earliest computers, freeing programmers from tedium such as remembering numeric codes and calculating addresses. They were once widely used for all sorts of programming. However, by the 1980s (1990s on microcomputers), their use had largely been supplanted by high-level languages, in the search for improved programming productivity. Today, although assembly language is almost always handled and generated by compilers, it is still used for direct hardware manipulation, access to specialized processor instructions, or to address critical performance issues. Typical uses are device drivers, low-level embedded systems, and real-time systems.

Historically, a large number of programs have been written entirely in assembly language. Operating systems were almost exclusively written in assembly language until the widespread acceptance of C in the 1970s and early 1980s. Many commercial applications were written in assembly language as well, including a large amount of the IBM mainframe software written by large corporations. COBOL and FORTRAN eventually displaced much of this work, although a number of large organizations retained assembly-language application infrastructures well into the 90s.

Most early microcomputers relied on hand-coded assembly language, including most operating systems and large applications. This was because these systems had severe resource constraints, imposed idiosyncratic memory and display architectures, and provided limited, buggy system services. Perhaps more important was the lack of first-class high-level language compilers suitable for microcomputer use. A psychological factor may have also played a role: the first generation of microcomputer programmers retained a hobbyist, "wires and pliers" attitude.

In a more commercial context, the biggest reasons for using assembly language were minimal bloat (size), minimal overhead, greater speed, and reliability.

Typical examples of large assembly language programs from this time are IBM PC DOS operating systems and early applications such as the spreadsheet program Lotus 1-2-3, and almost all popular games for the Atari 800 family of home computers. Even into the 1990s, most console video games were written in assembly, including most games for the Mega Drive/Genesis and the Super Nintendo Entertainment System. According to some industry insiders, the assembly language was the best computer language to use to get the best performance out of the Sega Saturn, a console that was notoriously challenging to develop and program games for. The popular arcade game NBA Jam (1993) is another example. On the Commodore 64, Amiga, Atari ST, as well as ZX Spectrum home computers, assembler has long been the primary development language. This was in large

part because BASIC dialects on these systems offered insufficient execution speed, as well as insufficient facilities to take full advantage of the available hardware on these systems. Some systems, most notably Amiga, even have IDEs with highly advanced debugging and macro facilities, such as the freeware ASM-One assembler, comparable to that of Microsoft Visual Studio facilities (ASM-One predates Microsoft Visual Studio).

The Assembler for the VIC-20 was written by Don French and published by *French Silk*. At 1639 bytes in length, its author believes it is the smallest symbolic assembler ever written. The assembler supported the usual symbolic addressing and the definition of character strings or hex strings. It also allowed address expressions which could be combined with addition, subtraction, multiplication, division, logical AND, logical OR, and exponentiation operators.

Current usage

There have always been debates over the usefulness and performance of assembly language relative to high-level languages. Assembly language has specific niche uses where it is important; see below. But in general, modern optimizing compilers are claimed to render high-level languages into code that can run as fast as hand-written assembly, despite the counter-examples that can be found. The complexity of modern processors and memory sub-system makes effective optimization increasingly difficult for compilers, as well as assembly programmers. Moreover, and to the dismay of efficiency lovers, increasing processor performance has meant that most CPUs sit idle most of the time, with delays caused by predictable bottlenecks such as I/O operations and paging. This has made raw code execution speed a non-issue for many programmers.

There are some situations in which practitioners might choose to use assembly language, such as when:

- a stand-alone binary executable is required, i.e. one that must execute without recourse to the run-time components or libraries associated with a high-level language; this is perhaps the most common situation. These are embedded programs that store only a small amount of memory and the device is intended to do single purpose tasks. Such examples consist of telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, and sensors.
- interacting directly with the hardware, for example in device drivers and interrupt handlers.
- using processor-specific instructions not exploited by or available to the compiler. A common example is the bitwise rotation instruction at the core of many encryption algorithms.
- creating vectorized functions for programs in higher-level languages such as C. In the higher-level language this is sometimes aided by compiler intrinsic functions which map directly to SIMD mnemonics, but nevertheless result in a one-to-one assembly conversion specific for the given vector processor.
- extreme optimization is required, e.g., in an inner loop in a processor-intensive algorithm. Game programmers take advantage of the abilities of hardware

features in systems, enabling games to run faster. Also large scientific simulations require highly optimized algorithms, e.g. linear algebra with BLAS or discrete cosine transformation (e.g. SIMD assembly version from x264)
- a system with severe resource constraints (e.g., an embedded system) must be hand-coded to maximize the use of limited resources; but this is becoming less common as processor price decreases and performance improves.
- no high-level language exists, on a new or specialized processor, for example.
- writing real-time programs that need precise timing and responses, such as simulations, flight navigation systems, and medical equipment. For example, in a fly-by-wire system, telemetry must be interpreted and acted upon within strict time constraints. Such systems must eliminate sources of unpredictable delays, which may be created by (some) interpreted languages, automatic garbage collection, paging operations, or preemptive multitasking. However, some higher-level languages incorporate run-time components and operating system interfaces that can introduce such delays. Choosing assembly or lower-level languages for such systems gives programmers greater visibility and control over processing details.
- complete control over the environment is required, in extremely high security situations where nothing can be taken for granted.
- writing computer viruses, bootloaders, certain device drivers, or other items very close to the hardware or low-level operating system.
- writing instruction set simulators for monitoring, tracing and debugging where additional overhead is kept to a minimum
- reverse-engineering existing binaries that may or may not have originally been written in a high-level language, for example when cracking copy protection of proprietary software.
- reverse engineering and modifying video games (also termed ROM hacking), which is possible via several methods. The most widely employed is altering program code at the assembly language level.
- writing self modifying code, to which assembly language lends itself well.
- writing games and other software for graphing calculators.
- writing compiler software that generates assembly code, and the writers should therefore be expert assembly language programmers themselves.
- writing cryptographic algorithms that must always take strictly the same time to execute, preventing timing attacks.

Assembly language is still taught in most computer science and electronic engineering programs even though few programmers today regularly work with assembly language as a tool, the underlying concepts remain very important. Such fundamental topics as binary arithmetic, memory allocation, stack processing, character set encoding, interrupt processing, and compiler design would be hard to study in detail without a grasp of how a computer operates at the hardware level. Since a computer's behavior is fundamentally defined by its instruction set, the logical way to learn such concepts is to study an assembly language. Most modern computers have similar instruction sets. Therefore, studying a single assembly language is sufficient to learn: I) the basic concepts; II) to

recognize situations where the use of assembly language might be appropriate; and III) to see how efficient executable code can be created from high-level languages.

Typical applications

Hard-coded assembly language is typically used in a system's boot ROM (BIOS on IBM-compatible PC systems). This low-level code is used, among other things, to initialize and test the system hardware prior to booting the OS, and is stored in ROM. Once a certain level of hardware initialization has taken place, execution transfers to other code, typically written in higher level languages; but the code running immediately after power is applied is usually written in assembly language. The same is true of most boot loaders.

Many compilers render high-level languages into assembly first before fully compiling, allowing the assembly code to be viewed for debugging and optimization purposes. Relatively low-level languages, such as C, often provide special syntax to embed assembly language directly in the source code. Programs using such facilities, such as the Linux kernel, can then construct abstractions using different assembly language on each hardware platform. The system's portable code can then use these processor-specific components through a uniform interface.

Assembly language is also valuable in reverse engineering, since many programs are distributed only in machine code form, and machine code is usually easy to translate into assembly language and carefully examine in this form, but very difficult to translate into a higher-level language. Tools such as the Interactive Disassembler make extensive use of disassembly for such a purpose.

One niche that makes use of assembly language is the demoscene. Certain competitions require contestants to restrict their creations to a very small size (e.g. 256B, 1KB, 4KB or 64 KB), and assembly language is the language of choice to achieve this goal. When resources, especially CPU processing-constrained systems, like the earlier Amiga models, and the Commodore 64, are a concern, assembler coding is a must. Optimized assembler code is written "by hand" and instructions are sequenced manually by programmers in an attempt to minimize the number of CPU cycles used. The CPU constraints are so great that every CPU cycle counts. However, using such methods has enabled systems like the Commodore 64 to produce real-time 3D graphics with advanced effects, a feat which might be considered unlikely or even impossible for a system with a 1.02MHz processor.

Related terminology

- **Assembly language** or **assembler language** is commonly called **assembly**, **assembler**, **ASM**, or **symbolic machine code**. A generation of IBM mainframe programmers called it **BAL** for *Basic Assembly Language*.

  Note: Calling the language **assembler** is of course potentially confusing and ambiguous, since this is also the name of the utility program that translates assembly language statements into machine code. Some may regard this as

imprecision or error. However, this usage has been common among professionals and in the literature for decades. Similarly, some early computers called their *assembler* their **assembly program**.)

- The computational step where an assembler is run, including all macro processing, is termed **assembly time**.
- The use of the word **assembly** dates from the early years of computers (*cf.* short code, speedcode).
- A **cross assembler** is functionally just an assembler. This term is used to stress that the assembler is run on a different computer than the target system, the system on which the resulting code is run. Because nowadays assemblers are written portably in a high level language like C, this is largely irrelevant. Cross assembling may be necessary if the target system lacks the capacity to run an assembler itself. This is typically the case for small embedded systems. The most important distinguishing feature of a cross assembler is that it provides for or interfaces to facilities to transport the code to the target processor, e.g. to reside in flash or EPROM. It generates a binary image, or Intel HEX file rather than an object file.
- An **assembler directive** or *pseudo-opcode* is a command given to an assembler. These directives may do anything from telling the assembler to include other source files, to telling it to allocate memory for constant data.

List of assemblers for different computer architectures

The following page has a list of different assemblers for the different computer architectures, along with any associated information for that specific assembler:

- List of assemblers

Further details

For any given personal computer, mainframe, embedded system, and game console, both past and present, at least one — possibly dozens — of assemblers have been written.

On Unix systems, the assembler is traditionally called as, although it is not a single body of code, being typically written anew for each port. A number of Unix variants use GAS.

Within processor groups, each assembler has its own dialect. Sometimes, some assemblers can read another assembler's dialect, for example, TASM can read old MASM code, but not the reverse. FASM and NASM have similar syntax, but each support different macros that could make them difficult to translate to each other. The basics are all the same, but the advanced features will differ.

Also, assembly can sometimes be portable across different operating systems on the same type of CPU. Calling conventions between operating systems often differ slightly or not at all, and with care it is possible to gain some portability in assembly language, usually

by linking with a C library that does not change between operating systems. An instruction set simulator (which would ideally be written in an assembler language) can, in theory, process the object code/ binary of *any* assembler to achieve portability even across platforms (with an overhead no greater than a typical bytecode interpreter). This is essentially what microcode achieves when a hardware platform changes internally.

For example, many things in libc depend on the preprocessor to do OS-specific, C-specific things to the program before compiling. In fact, some functions and symbols are not even guaranteed to exist outside of the preprocessor. Worse, the size and field order of structs, as well as the size of certain typedefs such as off_t, are entirely unavailable in assembly language without help from a configure script, and differ even between versions of Linux, making it impossible to portably call functions in libc other than ones that only take simple integers and pointers as parameters. To address this issue, FASMLIB project provides a portable assembly library for Win32 and Linux platforms, but it is yet very incomplete.

Some higher level computer languages, such as C and Borland Pascal, support inline assembly where relatively brief sections of assembly code can be embedded into the high level language code. The Forth language commonly contains an assembler used in CODE words.

Many people use an emulator to debug their assembly-language programs.

Example listing of assembly language source code

Address	Label	Instruction (AT&T syntax)	Object code
		.begin	
		.org 2048	
	a_start	.equ 3000	
2048		ld length,%	
2064		be done	00000010 10000000 00000000 00000110
2068		addcc %r1,-4,%r1	10000010 10000000 01111111 11111100
2072		addcc %r1,%r2,%r4	10001000 10000000 01000000 00000010
2076		ld %r4,%r5	11001010 00000001 00000000 00000000
2080		ba loop	00010000 10111111 11111111 11111011
2084		addcc %r3,%r5,%r3	10000110 10000000 11000000 00000101
2088	done:	jmpl %r15+4,%r0	10000001 11000011 11100000 00000100
2092	length:	20	00000000 00000000 00000000 00010100

```
2096    address: a_start                    00000000 00000000 00001011
                                            10111000

                 .org a_start

3000    a:
```

Example of a selection of instructions (for a virtual computer) with the corresponding address in memory where each instruction will be placed. These addresses are not static, see memory management. Accompanying each instruction is the generated (by the assembler) object code that coincides with the virtual computer's architecture (or ISA).

Chapter 7

Machine Code

Machine code or **machine language** is a system of instructions and data executed directly by a computer's central processing unit. Machine code may be regarded as a primitive (and cumbersome) programming language or as the lowest-level representation of a compiled and/or assembled computer program. Programs in interpreted languages are **not** represented by machine code however, although their *interpreter* (which may be seen as a processor executing the higher level program) often is. Machine code is sometimes called **native code** when referring to platform-dependent parts of language features or libraries. Machine code should not be confused with so called "bytecode", which is executed by an interpreter.

Machine code instructions

Every processor or processor family has its own machine code instruction set. Instructions are patterns of bits that by physical design correspond to different commands to the machine. The instruction set is thus specific to a class of processors using (much) the same architecture. Successor or derivative processor designs often include all the instructions of a predecessor and may add additional instructions. Occasionally a successor design will discontinue or alter the meaning of some instruction code (typically because it is needed for new purposes), affecting code compatibility to some extent; even nearly completely compatible processors may show slightly different behavior for some instructions but this is seldom a problem. Systems may also differ in other details, such as memory arrangement, operating systems, or peripheral devices; because a program normally relies on such factors, different systems will typically not run the same machine code, even when the same type of processor is used.

A machine code instruction set may have all instructions of the same length, or it may have variable-length instructions. How the patterns are organized varies strongly with the particular architecture and often also with the type of instruction. Most instructions have one or more opcode fields which specifies the basic instruction type (such as arithmetic,

logical, jump, etc) and the actual operation (such as add or compare) and other fields that may give the type of the operand(s), the addressing mode(s), the addressing offset(s) or index, or the actual value itself (such constant operands contained in an instruction are called *immediates*).

Programs

A computer program is a sequence of instructions that are executed by a CPU. While simple processors execute instructions one after the other, superscalar processors are capable of executing several instructions at once.

Program flow may be influenced by special 'jump' instructions that transfer execution to an instruction other than the following one. Conditional jumps are taken (execution continues at another address) or not (execution continues at the next instruction) depending on some condition.

Assembly languages

A much more readable rendition of machine language, called assembly language, uses mnemonic codes to refer to machine code instructions, rather than simply using the instructions' numeric values. For example, on the Zilog Z80 processor, the machine code `00000101`, which causes the CPU to decrement the `B` processor register, would be represented in assembly language as `DEC B`.

Example

The MIPS architecture provides a specific example for a machine code whose instructions are always 32 bits long. The general type of instruction is given by the *op* (operation) field, the highest 6 bits. J-type (jump) and I-type (immediate) instructions are fully specified by *op*. R-type (register) instructions include an additional field *funct* to determine the exact operation. The fields used in these types are:

```
   6      5     5     5     5      6 bits
[  op  |  rs |  rt |  rd |shamt| funct]  R-type
[  op  |  rs |  rt | address/immediate]  I-type
[  op  |        target address        ]  J-type
```

rs, *rt*, and *rd* indicate register operands; *shamt* gives a shift amount; and the *address* or *immediate* fields contain an operand directly.

For example adding the registers 1 and 2 and placing the result in register 6 is encoded:

```
[  op  |  rs |  rt |  rd |shamt| funct]
    0     1     2     6     0     32     decimal
 000000 00001 00010 00110 00000 100000   binary
```

Load a value into register 8, taken from the memory cell 68 cells after the location listed in register 3:

```
[  op   |  rs |  rt | address/immediate]
   35     3     8           68           decimal
 100011 00011 01000 00000 00001 000100   binary
```

Jumping to the address 1024:

```
[  op   |        target address        ]
    2                 1024              decimal
 000010 00000 00000 00000 10000 000000   binary
```

Relationship to microcode

In some computer architectures, the machine code is implemented by a more fundamental underlying layer of programs called microprograms, providing a common machine language interface across a line or family of different models of computer with widely different underlying dataflows. This is done to facilitate porting of machine language programs between different models. An example of this use is the IBM System/360 family of computers and their successors. With dataflow path widths of 8 bits to 64 bits and beyond, they nevertheless present a common architecture at the machine language level across the entire line.

Using a microcode layer to implement an emulator enables the computer to present the architecture of an entirely different computer. The System/360 line used this to allow porting programs from earlier IBM machines to the new family of computers, e.g. an IBM 1401/1440/1460 emulator on the IBM S/360 model 40.

Storing in memory

The Harvard architecture is a computer architecture with physically separate storage and signal pathways for the code (instructions) and data. Today, most processors implement such separate signal pathways for performance reasons but actually implement a Modified Harvard architecture, so they can support tasks like loading a program from disk storage as data and then executing it. Harvard architecture is contrasted to the Von Neumann architecture, where data and code are stored in the same memory.

From the point of view of a process, the *code space* is the part of its address space where code in execution is stored. In multi-threading environment, threads share code space along with data space, which reduces the overhead of context switching considerably as compared to process switching.

Readability by humans

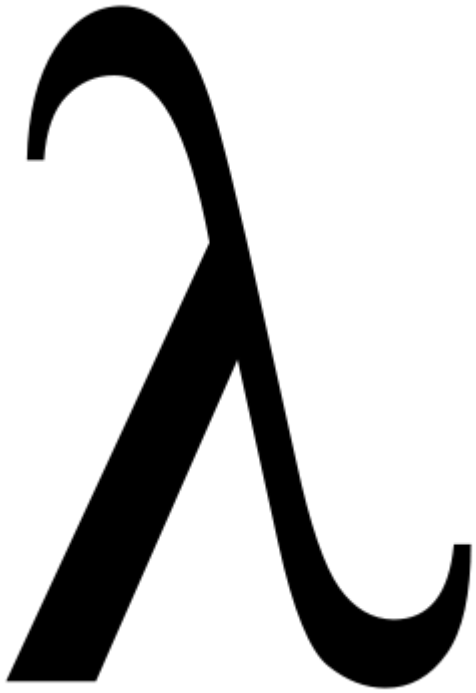
It has been said that machine code is so unreadable that the Copyright Office cannot even identify whether a particular encoded program is an original work of authorship.

Hofstadter writes, "Looking at a program written in machine language is vaguely comparable to looking at a DNA molecule atom by atom."

Chapter 8

Programming Language Theory & Language Primitive

Programming Language Theory



Lambda calculus is a formal system for function definition, function application and recursion introduced by Alonzo Church in the 1930s.

Programming language theory (commonly known as **PLT**) is a branch of computer science that deals with the design, implementation, analysis, characterization, and

classification of programming languages and their individual features. It falls within the discipline of computer science, both depending on and affecting mathematics, software engineering and linguistics. It is a well-recognized branch of computer science, and an active research area, with results published in numerous journals dedicated to PLT, as well as in general computer science and engineering publications. Most undergraduate computer science programs require coursework in the topic.

History

In some ways, the history of programming language theory predates even the development of programming languages themselves. The lambda calculus, developed by Alonzo Church and Stephen Cole Kleene in the 1930s, is considered by some to be the world's first programming language, even though it was intended to *model* computation rather than being a means for programmers to *describe* algorithms to a computer system. Many modern functional programming languages have been described as providing a "thin veneer" over the lambda calculus , and many are easily described in terms of it.

The first programming language to be proposed was Plankalkül, which was designed by Konrad Zuse in the 1940s, but not publicly known until 1972 (and not implemented until 1998). The first widely known and successful programming language was Fortran, developed from 1954 to 1957 by a team of IBM researchers led by John Backus. The success of FORTRAN led to the formation of a committee of scientists to develop a "universal" computer language; the result of their effort was ALGOL 58. Separately, John McCarthy of MIT developed the Lisp programming language (based on the lambda calculus), the first language with origins in academia to be successful. With the success of these initial efforts, programming languages became an active topic of research in the 1960s and beyond.

Some other key events in the history of programming language theory since then:

- In the 1950s, Noam Chomsky developed the Chomsky hierarchy in the field of linguistics; a discovery which has directly impacted programming language theory and other branches of computer science.
- In the 1960s
    - The Simula language was developed by Ole-Johan Dahl and Kristen Nygaard; it is widely considered to be the first example of an object-oriented programming language; Simula also introduced the concept of coroutines.
    - In 1964, Peter Landin is the first to realize Church's lambda calculus can be used to model programming languages. He introduces the SECD machine which "interprets" lambda expressions.
    - In 1965, Landin introduces the J operator, essentially a form of continuation.
    - In 1966, Landin introduces ISWIM, an abstract computer programming language in his article *The Next 700 Programming Languages*. It is

influential in the design of languages leading to the Haskell programming language.
    - In 1969, Tony Hoare introduces the Hoare logic, a form of axiomatic semantics.
    - In 1969, William Alvin Howard observed that a "high-level" proof system, referred to as natural deduction, can be directly interpreted in its intuitionistic version as a typed variant of the model of computation known as lambda calculus. This became known as the Curry–Howard correspondence.
- In the 1970s:
    - In 1970, Dana Scott first publishes his work on denotational semantics.
    - In 1972, Logic programming and Prolog were developed thus allowing computer programs to be expressed as mathematical logic.
    - In 1974, John C. Reynolds discovers System F. It had al ready been discovered in 1971 by the mathematical logician Jean-Yves Girard.
    - From 1975, Sussman and Steele develop the Scheme programming language, a Lisp dialect incorporating lexical scoping, a unified namespace, and elements from the Actor model including first-class continuations.
    - Backus, at the 1977 ACM Turing Award lecture, assailed the current state of industrial languages and proposed a new class of programming languages now known as function-level programming languages.
    - In 1977, Gordon Plotkin introduces Programming Computable Functions, an abstract typed functional language.
    - In 1978, Robin Milner introduces the Hindley–Milner type inference algorithm for the ML programming language. Type theory became applied as a discipline to programming languages, this application has led to tremendous advances in type theory over the years.
- In the 1980s:
    - A team of scientists at Xerox PARC led by Alan Kay develop Smalltalk, an object-oriented language widely known for its innovative development environment.
    - There emerged process calculi, such as the Calculus of Communicating Systems of Robin Milner, and the Communicating sequential processes model of C. A. R. Hoare, as well as similar models of concurrency such as the Actor model of Carl Hewitt.
    - In 1985, The release of Miranda sparks an academic interest in lazy-evaluated pure functional programming languages. A committee was formed to define an open standard resulting in the release of the Haskell 1.0 standard in 1990.
    - Bertrand Meyer created the methodology Design by contract and incorporated it into the Eiffel programming language.
- In the 1990s:
    - Gregor Kiczales, Jim Des Rivieres and Daniel G. Bobrow published the book The Art of the Metaobject Protocol.

- Eugenio Moggi and Philip Wadler introduced the use of monads for structuring programs written in functional programming languages.

Sub-disciplines and related fields

There are several fields of study which either lie within programming language theory, or which have a profound influence on it; many of these have considerable overlap. In addition, PLT makes use of many other branches of mathematics, including computability theory, category theory, and set theory.

Formal semantics

Formal semantics is the formal specification of the behaviour of computer programs and programming languages. Three common approaches to describe the semantics or "meaning" of a computer program are denotational semantics, operational semantics and axiomatic semantics.

Type theory

Type theory is the study of type systems; which are "tractable syntactic method(s) for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute." (Types and Programming Languages, MIT Press, 2002). Many programming languages are distinguished by the characteristics of their type systems.

Program analysis and transformation

Program analysis is the general problem of examining a program and determining key characteristics (such as the absence of classes of program errors). Program transformation is the process of transforming a program in one form (language) to another form.

Comparative programming language analysis

Comparative programming language analysis seeks to classify programming languages into different types based on their characteristics; broad categories of programming languages are often known as programming paradigms.

Generic and metaprogramming

Metaprogramming is the generation of higher-order programs which, when executed, produce programs (possibly in a different language, or in a subset of the original language) as a result.

Domain-specific languages

Domain-specific languages are languages constructed to efficiently solve problems in a particular problem domain.

Compiler construction

Compiler theory is the theory of writing *compilers* (or more generally, *translators*); programs which translate a program written in one language into another form. The actions of a compiler are traditionally broken up into *syntax analysis* (scanning and parsing), *semantic analysis* (determining what a program should do), *optimization* (improving the performance of a program as indicated by some metric; typically execution speed) and *code generation* (generation and output of an equivalent program in some target language; often the instruction set of a CPU).

Run-time systems

Runtime systems refers to the development of programming language runtime environments and their components, including virtual machines, garbage collection, and foreign function interfaces.

PLT-specific journals, publications, and conferences

Conferences are the primary venue for presenting research in programming languages. The most well known conferences include the Symposium on Principles of Programming Languages (POPL), Conference on Programming Language Design and Implementation (PLDI), the International Conference on Functional Programming (ICFP), and the International Conference on Object Oriented Programming, Systems, Languages and Applications (OOPSLA).

Notable journals that publish PLT research include the *ACM Transactions on Programming Languages and Systems*, *Journal of Functional Programming*, *Journal of Functional and Logic Programming*, and *Journal of Symbolic Computation*.

The Lambda *Symbol*

An unofficial symbol of the field of programming language theory is the lowercase Greek letter λ (lambda). This usage derives from the lambda calculus, a computational model widely used by programming language researchers. Many texts and papers on programming and programming languages utilize the lambda in some fashion. It graces the cover of the classic text *Structure and Interpretation of Computer Programs*, and the title of many of the so-called Lambda Papers, written by Gerald Jay Sussman and Guy Steele, the developers of the Scheme programming language. A popular website on programming language theory is called Lambda the Ultimate, in honor of Sussman and Steele's work.

Language Primitive

In computing, **language primitives** are the simplest elements available in a programming language. A primitive can be defined as the smallest 'unit of processing' available to a programmer of a particular machine, or can be an atomic element of an expression in a language.

Machine level primitives

A machine instruction, usually generated by an Assembler program, is often considered the smallest unit of processing although this is not always the case. It typically performs what is perceived to be one single operation such as copying a byte or string of bytes from one memory location to another or adding one processor register to another.

Micro code primitives

Many of today's computers, however, actually embody an even lower unit of processing known as microcode which interprets the "machine code" and it is then that the microcode instructions would be the *genuine* primitives. These instructions would typically be available for modification only by the hardware vendors programmers.

High level language primitives

A high-level programming language (*HLL*) program is composed of discrete statements and primitive data types that may also be *perceived* to perform a single operation or represent a single data item, but at a more abstract level than those provided by the machine. Copying a data item from one location to another may actually involve many machine instructions that, for instance,

- calculate the address of both operands in memory, based on their positions within a data structure,
- convert from one data type to another

before finally

- performing the final store operation to the target destination.

Some HLL statements, particularly those involving loops, can generate thousands or even millions of primitives in a low level language - which comprise the genuine instruction path length the processor has to execute at the lowest level. This perception has been referred to as the "Abstraction penalty"

Interpreted language primitives

An interpreted language statement has similarities to the HLL primitives but with a further added 'layer'. Before the statement can be executed in a manner very similar to a HLL statement, first, it has to be processed by an interpreter, a process that may involve many primitives in the target machine language.

Fourth and Fifth-generation programming language primitives

4gls and 5gls do not have a simple one-to-many correspondence from high-to-low level primitives. There are some elements of interpreted language primitives embodied in 4gl and 5gl specifications but the approach to the original problem is less a procedural language construct and are more oriented toward problem solving and systems engineering.

Chapter 9

Syntax (programming languages)

```
def add5(x):
   return x+5

def dotwrite(ast):
   nodename = getNodename()
   label=symbol.sym_name.get(int(ast[0]),ast[0])
   print '    %s [label="%s' % (nodename, label),
   if isinstance(ast[1], str):
      if ast[1].strip():
         print '= %s"];' % ast[1]
      else:
         print '"]'
    else:
      print '"];'
      children = []
      for  in n, childenumerate(ast[1:]):
         children.append(dotwrite(child))
      print ,'    %s -> {' % nodename
      for  in :namechildren
         print '%s' % name,
```

Syntax highlighting is often used to aid programmers in recognizing elements of source code. The language above is Python.

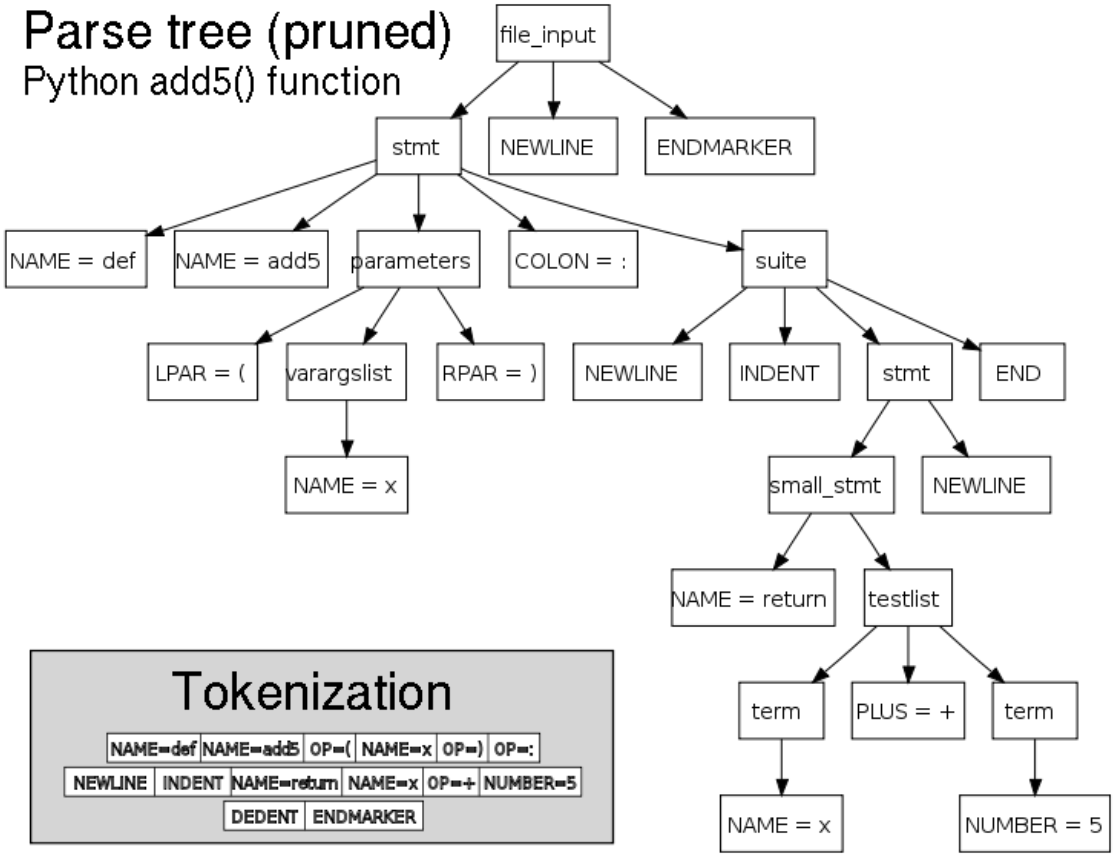
In computer science, the **syntax** of a programming language is the set of rules that define the combinations of symbols that are considered to be correctly structured programs in that language. The syntax of a language defines its surface form. Text-based programming languages are based on sequences of characters, while visual programming

languages are based on the spatial layout and connections between symbols (which may be textual or graphical).

The lexical grammar of a textual language specifies how characters must be chunked into tokens. Other syntax rules specify the permissible sequences of these tokens and the process of assigning meaning to these token sequences is part of semantics.

The syntactic analysis of source code usually entails the transformation of the linear sequence of tokens into a hierarchical syntax tree (abstract syntax trees are one convenient form of syntax tree). This process is called *parsing*, as it is in syntactic analysis in linguistics. Tools have been written that automatically generate parsers from a specification of a language grammar written in Backus-Naur form, e.g., Yacc (yet another compiler compiler).

Syntax definition



Parse tree of Python code with inset tokenization

The syntax of textual programming languages is usually defined using a combination of regular expressions (for lexical structure) and Backus-Naur Form (for grammatical structure) to inductively specify syntactic categories (nonterminals) and *terminal* symbols. Syntactic categories are defined by rules called *productions*, which specify the

values that belong to a particular syntactic category. Terminal symbols are the concrete characters or strings of characters (for example keywords such as *define*, *if*, *let*, or *void*) from which syntactically valid programs are constructed.

Below is a simple grammar, based on Lisp, which defines productions for the syntactic categories *expression*, *atom*, *number*, *symbol*, and *list*:

```
expression ::= atom | list
atom  ::= number | symbol
number  ::= [+-]?['0'-'9']+
symbol  ::= ['A'-'Z''a'-'z'].*
list  ::= '(' expression* ')'
```

This grammar specifies the following:

- an *expression* is either an *atom* or a *list*;
- an *atom* is either a *number* or a *symbol*;
- a *number* is an unbroken sequence of one or more decimal digits, optionally preceded by a plus or minus sign;
- a *symbol* is a letter followed by zero or more of any characters (excluding whitespace); and
- a *list* is a matched pair of parentheses, with zero or more *expressions* inside it.

Here the decimal digits, upper- and lower-case characters, and parentheses are terminal symbols.

The following are examples of well-formed token sequences in this grammar: '`12345`', '`()`','`(a b c232 (1))`'

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars. However, there are exceptions. In some languages like Perl and Lisp the specification (or implementation) of the language allows constructs that execute during the parsing phase. Furthermore, these languages have constructs that allow the programmer to alter the behavior of the parser. This combination effectively blurs the distinction between parsing and execution, and makes syntax analysis an undecidable problem in these languages, meaning that the parsing phase may not finish. For example, in Perl it is possible to execute code during parsing using a `BEGIN` statement, and Perl function prototypes may alter the syntactic interpretation, and possibly even the syntactic validity of the remaining code. Similarly, Lisp macros introduced by the `defmacro` syntax also execute during parsing, meaning that a Lisp compiler must have an entire Lisp run-time system present. In contrast C macros are merely string replacements, and do not require code execution.

Syntax versus semantics

The syntax of a language describes the form of a valid program, but does not provide any information about the meaning of the program or the results of executing that program. The meaning given to a combination of symbols is handled by semantics (either formal or hard-coded in a reference implementation). Not all syntactically correct programs are semantically correct. Many syntactically correct programs are nonetheless ill-formed, per the language's rules; and may (depending on the language specification and the soundness of the implementation) result in an error on translation or execution. In some cases, such programs may exhibit undefined behavior. Even when a program is well-defined within a language, it may still have a meaning that is not intended by the person who wrote it.

Using natural language as an example, it may not be possible to assign a meaning to a grammatically correct sentence or the sentence may be false:

- "Colorless green ideas sleep furiously." is grammatically well-formed but has no generally accepted meaning.
- "John is a married bachelor." is grammatically well-formed but expresses a meaning that cannot be true.

The following C language fragment is syntactically correct, but performs an operation that is not semantically defined (because `p` is a null pointer, the operations `p->real` and `p->im` have no meaning):

```
complex *p = NULL;
complex abs_p = sqrt (p->real * p->real + p->im * p->im);
```

Chapter 10

Type System

In computer science, a **type system** may be defined as "a tractable syntactic framework for classifying phrases according to the kinds of values they compute". A type system associates *types* with each computed value. By examining the flow of these values, a type system attempts to prove that no *type errors* can occur. The type system in question determines what constitutes a type error, but a type system generally seeks to guarantee that operations expecting a certain kind of value are not used with values for which that operation does not make sense.

A compiler may use the static type of a value to optimize the storage it needs and the choice of algorithms for operations on the value. In many C compilers the "float" data type, for example, is represented in 32 bits, in accord with the IEEE specification for single-precision floating point numbers. C thus uses floating-point-specific operations on those values (floating-point addition, multiplication, etc.).

The depth of type constraints and the manner of their evaluation affect the *typing* of the language. A programming language may further associate an operation with varying concrete algorithms on each type in the case of type polymorphism. Type theory is the study of type systems, although the concrete type systems of programming languages originate from practical issues of computer architecture, compiler implementation, and language design.

Fundamentals

Assigning data types (*typing*) gives meaning to sequences of bits. Types usually have associations either with values in memory or with objects such as variables. Because any value simply consists of a sequence of bits in a computer, hardware makes no intrinsic distinction even between memory addresses, instruction code, characters, integers and floating-point numbers, being unable to discriminate between them based on bit pattern alone. Associating a sequence of bits and a type informs programs and programmers how that sequence of bits should be understood.

Major functions provided by type systems include:

- *Safety* – Use of types may allow a compiler to detect meaningless or probably invalid code. For example, we can identify an expression `3 / "Hello, World"` as invalid because the rules of arithmetic do not specify how to divide an integer by a string. As discussed below, strong typing offers more safety, but generally does not guarantee complete safety.
- *Optimization* – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of 4 bytes, the compiler may be able to use more efficient machine instructions.
- *Documentation* – In more expressive type systems, types can serve as a form of documentation, since they can illustrate the intent of the programmer. For instance, timestamps may be represented as integers—but if a programmer declares a function as returning a timestamp type rather than merely an integer type, this documents part of the meaning of the function.
- *Abstraction* (or *modularity*) – Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can think of a string as a collection of character values instead of as a mere array of bytes. Or, types can allow programmers to express the interface between two subsystems. This helps localize the definitions required for interoperability of the subsystems and prevents inconsistencies when those subsystems communicate.

Type safety contributes to program correctness, but cannot guarantee it unless the type checking itself becomes an undecidable problem. Depending on the specific type system, a program may give the wrong result and be safely typed, producing no compiler errors. For instance, division by zero is not caught by the type checker in most programming languages; instead it is a runtime error. To prove the absence of more general defects, other kinds of formal methods, collectively known as program analyses, are in common use, as well as software testing, a widely used empirical method for finding errors that the type checker cannot detect.

A program typically associates each value with one particular type (although a type may have more than one subtype). Other entities, such as objects, modules, communication channels, dependencies, or even types themselves, can become associated with a type. Some implementations might make the following identifications (though these are technically different concepts):

- data type – a type of a value
- class – a type of an object
- kind – a type of a type

A *type system*, specified for each programming language, controls the ways typed programs may behave, and makes behavior outside these rules illegal. An *effect system* typically provides more fine-grained control than does a type system.

Formally, type theory studies type systems. More elaborate type systems (such as dependent types) allow for finer-grained program specifications to be verified by a type checker, but this comes at a price, as type inference and other properties generally become undecidable, and type checking itself is dependent on user-supplied proofs. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in type safe manner. As Mark Manasse concisely put it:

The fundamental problem addressed by a type theory is to insure that programs have meaning. The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them. The quest for richer type systems results from this tension.

Type checking

The process of verifying and enforcing the constraints of types – *type checking* – may occur either at compile-time (a static check) or run-time (a dynamic check). If a language specification requires its typing rules strongly (i.e., more or less allowing only those automatic type conversions which do not lose information), one can refer to the process as *strongly typed*, if not, as *weakly typed*. The terms are not used in a strict sense.

Static typing

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Statically typed languages include Ada, ActionScript 3, C, C++, C#, Eiffel, F#, Fortran, Go, Haskell, JADE, Java, ML, Objective-C, Ocaml, Pascal, Perl (with respect to distinguishing scalars, arrays, hashes and subroutines) and Scala. Static typing is a limited form of program verification: accordingly, 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 type checks every time the program is executed. Program execution may also be made more efficient (i.e. faster or taking reduced memory) by omitting runtime type checks and enabling other optimizations.

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 behaviour 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. Without static type checking,

even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors because the combination of all places where values are created and all places where a certain value is used must be taken into account.

The most widely used statically typed languages are not formally type safe. They have "loopholes" in the programming language specification enabling programmers to write code that circumvents the verification performed by a static type checker and so address a wider range of problems. For example, most C-style languages have type punning, and Haskell has such features as `unsafePerformIO`: such operations may be unsafe at runtime, in that they can cause unwanted behaviour due to incorrect typing of values when the program runs.

Dynamic typing

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 dynamic typing, values have types but variables do not; that is, a variable can refer to a value of any type.

Implementations of dynamically-typed languages generally associate run-time objects with "tags" containing their type information. This run-time classification is then used to implement type checks and dispatch overloaded functions, but can also enable pervasive uses of dynamic dispatch, late binding and similar idioms which would be cumbersome at best in a statically-typed language, requiring the use of variant types or similar features.

More broadly, as explained below, dynamic typing can improve support for dynamic programming language features, such as generating types and functionality based on run-time data. (Nevertheless, dynamically typed languages need not support any or all such features, and some *dynamic programming languages* are statically typed.) On the other hand, dynamic typing provides fewer *a priori* guarantees: a dynamically typed language accepts and attempts to execute some programs which would be ruled as invalid by a static type checker, either due to errors in the program or due to static type checking being too conservative.

Dynamic typing may result in runtime type errors—that is, at runtime, a value may have an unexpected type, and an operation nonsensical for that type is applied. Such errors may occur long after the place where the programming mistake was made—that is, the place where the wrong type of data passed into a place it should not have. This may make the bug difficult to locate.

Dynamically typed language systems' run-time checks can potentially be more sophisticated than those of statically typed languages, as they can use dynamic information as well as any information from the source code. On the other hand, runtime checks only assert that conditions hold in a particular execution of the program, and the checks are repeated for every execution of the program.

Dynamically typed languages include APL, Erlang, Groovy, JavaScript, Lisp, Lua, MATLAB/GNU Octave, Perl (with respect to user-defined types but not built-in types), PHP, Prolog, Python, Ruby, Smalltalk and Tcl.

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. In practice, the testing done to ensure correct program operation can detect a much wider range of errors than static type-checking, but conversely cannot search as comprehensively for the errors that static type checking is able to detect.

Combinations of dynamic and static typing

The presence of static typing in a programming language does not necessarily imply the absence of all dynamic typing mechanisms. For example, Java and some other ostensibly statically typed languages, support downcasting and other type operations which depend on runtime type checks, a form of dynamic typing. More generally, most programming languages include mechanisms for dispatching over different 'kinds' of data, such as disjoint unions, variant types and polymorphic objects: Even when not interacting with type annotations or type checking, such mechanisms are materially similar to dynamic typing implementations.

Certain languages, for example Clojure, are dynamically typed by default but allow this behaviour to be overridden through the use of explicit type hints that result in static typing. One reason to use such hints would be to achieve the performance benefits of static typing in performance-sensitive parts of code.

As of the 4.0 Release, the .NET Framework supports a variant of dynamic typing via the System.Dynamic namespace whereby a *static* object of type 'dynamic' is a placeholder for the .NET runtime to interrogate its dynamic facilities to resolve the object reference.

Static and dynamic type checking in practice

The choice between static and dynamic typing requires trade-offs.

Static typing can find type errors reliably at compile time. This should increase the reliability of the delivered program. However, programmers disagree over how commonly type errors occur, and thus disagree over the proportion of those bugs which are coded that would be caught by appropriately representing the designed types in code. Static typing advocates believe programs are more reliable when they have been well type-checked, while dynamic typing advocates point to distributed code that has proven reliable and to small bug databases. The value of static typing, then, presumably increases as the strength of the type system is increased. Advocates of dependently typed languages such as Dependent ML and Epigram have suggested that almost all bugs can be considered type errors, if the types used in a program are properly declared by the programmer or correctly inferred by the compiler.

Static typing usually results in compiled code that executes more quickly. When the compiler knows the exact data types that are in use, it can produce optimized machine code. Further, compilers for statically typed languages can find assembler shortcuts more easily. Some dynamically typed languages such as Common Lisp allow optional type declarations for optimization for this very reason. Static typing makes this pervasive.

By contrast, dynamic typing may allow compilers to run more quickly and allow interpreters to dynamically load new code, since changes to source code in dynamically typed languages may result in less checking to perform and less code to revisit. This too may reduce the edit-compile-test-debug cycle.

Statically typed languages which lack type inference (such as Java and C) require that programmers declare the types they intend a method or function to use. This can serve as additional documentation for the program, which the compiler will not permit the programmer to ignore or permit to drift out of synchronization. However, a language can be statically typed without requiring type declarations (examples include Haskell, Scala and to a lesser extent C#), so explicit type declaration is not a necessary requirement for static typing in all languages.

Dynamic typing allows constructs that some static type checking would reject as illegal. For example, *eval* functions, which execute arbitrary data as code, become possible. Furthermore, dynamic typing better accommodates transitional code and prototyping, such as allowing a placeholder data structure (mock object) to be transparently used in place of a full-fledged data structure (usually for the purposes of experimentation and testing).

Dynamic typing is used in duck typing which can support easier code reuse.

Dynamic typing typically makes metaprogramming more effective and easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code. More advanced run-time constructs such as metaclasses and introspection are often more difficult to use in statically typed languages. In some languages, such features may also be used e.g. to generate new types and behaviors on the fly, based on run-time data. Such advanced constructs are often provided by dynamic programming languages; many of these are dynamically typed, although *dynamic typing* need not be related to *dynamic programming languages*.

Strong and weak typing

A type system is said to feature strong typing when it specifies one or more restrictions on how operations involving values of different data types can be intermixed. A computer language that implements strong typing will prevent the successful execution of an operation on arguments which have the wrong type.

Weak typing means that a language implicitly converts (or casts) types when used. Consider the following example:

```
var x := 5;     // (1)  (x is an integer)
var y := "37"; // (2)  (y is a string)
x + y;          // (3)  (?)
```

In a weakly typed language, the result of this operation is unclear. Some languages, such as Visual Basic, would produce runnable code producing the result 42: the system would convert the string "37" into the number 37 to forcibly make sense of the operation. Other languages like JavaScript would produce the result "537": the system would convert the number 5 to the string "5" and then concatenate the two. In both Visual Basic and JavaScript, the resulting type is determined by rules that take both operands into consideration. In some languages, such as AppleScript, the type of the resulting value is determined by the type of the left-most operand only.

A C cast gone wrong exemplifies the problem that can occur if strong typing is absent; if a programmer casts a value from one type to another in C, not only must the compiler allow the code at compile time, but the runtime must allow it as well. This may permit more compact and faster C code, but it can make debugging more difficult.

Safely and unsafely typed systems

A third way of categorizing the type system of a programming language uses the safety of typed operations and conversions. Computer scientists consider a language "type-safe" if it does not allow operations or conversions which lead to erroneous conditions.

Some observers use the term *memory-safe language* (or just *safe language*) to describe languages that do not allow undefined operations to occur. For example, a memory-safe language will check array bounds, or else statically guarantee (i.e., at compile time before execution) that array accesses out of the array boundaries will cause compile-time and perhaps runtime errors.

```
var x := 5;      // (1)
var y := "37";  // (2)
var z := x + y; // (3)
```

In languages like Visual Basic, variable z in the example acquires the value 42. While the programmer may or may not have intended this, the language defines the result specifically, and the program does not crash or assign an ill-defined value to z. In this respect, such languages are type-safe; however, if the value of y was a string that could not be converted to a number (e.g. "hello world"), the results would be undefined. Such languages are type-safe (in that they will not crash) but can easily produce undesirable results.

Now let us look at the same example in C:

```
int x = 5;
char y[] = "37";
char* z = x + y;
```

In this example z will point to a memory address five characters beyond y, equivalent to three characters after the terminating zero character of the string pointed to by y. The content of that location is undefined, and might lie outside addressable memory. The mere computation of such a pointer may result in undefined behavior (including the program crashing) according to C standards, and in typical systems dereferencing z at this point could cause the program to crash. We have a well-typed, but not memory-safe program—a condition that cannot occur in a type-safe language.

Polymorphism and types

The term "polymorphism" refers to the ability of code (in particular, methods or classes) to act on values of multiple types, or to the ability of different instances of the same data-structure to contain elements of different types. Type systems that allow polymorphism generally do so in order to improve the potential for code re-use: in a language with polymorphism, programmers need only implement a data structure such as a list or an associative array once, rather than once for each type of element with which they plan to use it. For this reason computer scientists sometimes call the use of certain forms of polymorphism *generic programming*. The type-theoretic foundations of polymorphism are closely related to those of abstraction, modularity and (in some cases) subtyping.

Duck typing

In "duck typing", a statement calling a method *m* on an object does not rely on the declared type of the object; only that the object, of whatever type, must implement the method called. One way of looking at this is that in duck typing systems the type of an object is intrinsic to the object and is determined by what methods it implements, and hence that a duck typing system is by definition type-safe since one can only invoke operations an object actually implements. Another way of looking at this is that the object is a member of *several* types, including a type that describes the fact that it "has a method *m*." Type checking however occurs only on demand at runtime, every time the method *m* needs to be executed, not at compile-time or load-time.

Duck typing differs from structural typing in that, if the *part* (of the whole module structure) needed for a given local computation is present *at runtime*, the duck type system is satisfied in its type identity analysis. On the other hand, a structural type system would require the analysis of the whole module structure at compile-time to determine type identity or type dependence.

Duck typing differs from a nominative type system in a number of aspects. The most prominent ones are that, for duck typing, type information is determined at runtime (as contrasted to compile-time) and the name of the type is irrelevant to determine type identity or type dependence; only partial structure information is required for that, for a given point in the program execution.

Duck typing uses the premise that (referring to a value) "if it walks like a duck, and quacks like a duck, then it is a duck" (this is a reference to the duck test which is

attributed to James Whitcomb Riley). The term may have been coined by Alex Martelli in a 2000 message to the comp.lang.python newsgroup.

Specialized type systems

Many type systems have been created that are specialized for use in certain environments, with certain types of data, or for out-of-band static program analysis. Frequently these are based on ideas from formal type theory and are only available as part of prototype research systems.

Dependent types

Dependent types are based on the idea of using scalars or values to more precisely describe the type of some other value. For example, "matrix(3,3)" might be the type of a 3×3 matrix. We can then define typing rules such as the following rule for matrix multiplication:

$$\text{matrix\_multiply} : \text{matrix}(k,m) \times \text{matrix}(m,n) \rightarrow \text{matrix}(k,n)$$

where k, m, n are arbitrary positive integer values. A variant of ML called Dependent ML has been created based on this type system, but because type-checking conventional dependent types is undecidable, not all programs using them can be type-checked without some kind of limits. Dependent ML limits the sort of equality it can decide to Presburger arithmetic. Other languages such as Epigram make the value of all expressions in the language decidable so that type checking can be decidable, it is also possible to make the language Turing complete at the price of undecidable type checking like in Cayenne.

Linear types

Linear types, based on the theory of linear logic, and closely related to uniqueness types, are types assigned to values having the property that they have one and only one reference to them at all times. These are valuable for describing large immutable values such as strings, files, and so on, because any operation that simultaneously destroys a linear object and creates a similar object (such as '`str = str + "a"`') can be optimized "under the hood" into an in-place mutation. Normally this is not possible because such mutations could cause side effects on parts of the program holding other references to the object, violating referential transparency. They are also used in the prototype operating system Singularity for interprocess communication, statically ensuring that processes cannot share objects in shared memory in order to prevent race conditions. The Clean language (a Haskell-like language) uses this type system in order to gain a lot of speed while remaining safe.

Intersection types

Intersection types are types describing values that belong to *both* of two other given types with overlapping value sets. For example, in most implementations of C the signed char

has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting *either* signed or unsigned chars, because it is compatible with both types.

Intersection types are useful for describing overloaded function types: For example, if "int → int" is the type of functions taking an integer argument and returning an integer, and "float → float" is the type of functions taking a float argument and returning a float, then the intersection of these two types can be used to describe functions that do one or the other, based on what type of input they are given. Such a function could be passed into another function expecting an "int → int" function safely; it simply would not use the "float → float" functionality.

In a subclassing hierarchy, the intersection of a type and an ancestor type (such as its parent) is the most derived type. The intersection of sibling types is empty.

The Forsythe language includes a general implementation of intersection types. A restricted form is refinement types.

Union types

Union types are types describing values that belong to *either* of two types. For example, in C, the signed char has range -128 to 127, and the unsigned char has range 0 to 255, so the union of these two types would have range -128 to 255. Any function handling this union type would have to deal with integers in this complete range. More generally, the only valid operations on a union type are operations that are valid on *both* types being unioned. C's "union" concept is similar to union types, but is not typesafe because it permits operations that are valid on *either* type, rather than *both*. Union types are important in program analysis, where they are used to represent symbolic values whose exact nature (e.g., value or type) is not known.

In a subclassing hierarchy, the union of a type and an ancestor type (such as its parent) is the ancestor type. The union of sibling types is a subtype of their common ancestor (that is, all operations permitted on their common ancestor are permitted on the union type, but they may also have other valid operations in common).

Existential types

Existential types are frequently used in connection with record types to represent modules and abstract data types, due to their ability to separate implementation from interface. For example, the type "T = $\exists$ X { a: X; f: (X → int); }" describes a module interface that has a data member of type *X* and a function that takes a parameter of the *same* type *X* and returns an integer. This could be implemented in different ways; for example:

- intT = { a: int; f: (int → int); }
- floatT = { a: float; f: (float → int); }

These types are both subtypes of the more general existential type T and correspond to concrete implementation types, so any value of one of these types is a value of type T. Given a value "t" of type "T", we know that "t.f(t.a)" is well-typed, regardless of what the abstract type *X* is. This gives flexibility for choosing types suited to a particular implementation while clients that use only values of the interface type—the existential type—are isolated from these choices.

In general it's impossible for the typechecker to infer which existential type a given module belongs to. In the above example intT { a: int; f: (int → int); } could also have the type $\exists$X { a: X; f: (int → int); }. The simplest solution is to annotate every module with its intended type, e.g.:

- intT = { a: int; f: (int → int); } **as** $\exists$X { a: X; f: (X → int); }

Although abstract data types and modules had been implemented in programming languages for quite some time, it wasn't until 1988 that John C. Mitchell and Gordon Plotkin established the formal theory under the slogan: "Abstract [data] types have existential type". The theory is a second-order typed lambda calculus similar to System F, but with existential instead of universal quantification.

Explicit or implicit declaration and inference

Many static type systems, such as those of C and Java, require *type declarations*: The programmer must explicitly associate each variable with a particular type. Others, such as Haskell's, perform *type inference*: The compiler draws conclusions about the types of variables based on how programmers use those variables. For example, given a function *f(x,y)* which adds *x* and *y* together, the compiler can infer that *x* and *y* must be numbers – since addition is only defined for numbers. Therefore, any call to *f* elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error.

Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression `3.14` might imply a type of floating-point, while `[1, 2, 3]` might imply a list of integers – typically an array.

Type inference is in general possible if it is decidable in the type theory in question. Moreover, even if inference is undecidable in general for a given type theory, inference is often possible for a large subset of real-world programs. Haskell's type system, a version of Hindley-Milner, is a restriction of System Fω to so-called rank-1 polymorphic types, in which type inference is decidable. Most Haskell compilers allow arbitrary-rank polymorphism as an extension, but this makes type inference undecidable. (Type checking is decidable, however, and rank-1 programs still have type inference; higher rank polymorphic programs are rejected unless given explicit type annotations.)

Types of types

A *type of types* is a kind. Kinds appear explicitly in typeful programming, such as a *type constructor* in the Haskell language.

Types fall into several broad categories:

- Primitive types – the simplest kind of type; e.g., integer and floating-point number
  - Boolean
  - Integral types – types of whole numbers; e.g., integers and natural numbers
  - Floating point types – types of numbers in floating-point representation
- Reference types
- Option types
  - Nullable types
- Composite types – types composed of basic types; e.g., arrays or records.

  Abstract data types

- Algebraic types
- Subtype
- Derived type
- Object types; e.g., type variable
- Partial type
- Recursive type
- Function types; e.g., binary functions
- universally quantified types, such as parameterized types
- existentially quantified types, such as modules
- Refinement types – types which identify subsets of other types
- Dependent types – types which depend on terms (values)
- Ownership types – types which describe or constrain the structure of object-oriented systems

Compatibility: equivalence and subtyping

A type-checker for a statically typed language must verify that the type of any expression is consistent with the type expected by the context in which that expression appears. For instance, in an assignment statement of the form `x := e`, the inferred type of the expression *e* must be consistent with the declared or inferred type of the variable `x`. This notion of consistency, called *compatibility*, is specific to each programming language.

If the type of *e* and the type of `x` are the same and assignment is allowed for that type, then this is a valid expression. In the simplest type systems, therefore, the question of whether two types are compatible reduces to that of whether they are *equal* (or *equivalent*). Different languages, however, have different criteria for when two type expressions are understood to denote the same type. These different *equational theories*

of types vary widely, two extreme cases being *structural type systems,* in which any two types are equivalent that describe values with the same structure, and *nominative type systems,* in which no two syntactically distinct type expressions denote the same type (*i.e.,* types must have the same "name" in order to be equal).

In languages with subtyping, the compatibility relation is more complex. In particular, if *A* is a subtype of *B*, then a value of type *A* can be used in a context where one of type *B* is expected, even if the reverse is not true. Like equivalence, the subtype relation is defined differently for each programming language, with many variations possible. The presence of parametric or ad hoc polymorphism in a language may also have implications for type compatibility.

Programming style

Some programmers prefer statically typed languages; others prefer dynamically typed languages. Statically typed languages alert programmers to type errors during compilation, and they may perform better at runtime. Advocates of dynamically typed languages claim they better support rapid prototyping and that type errors are only a small subset of errors in a program. Likewise, there is often no need to manually declare all types in statically typed languages with type inference; thus, the need for the programmer to explicitly specify types of variables is automatically lowered for such languages; and some dynamic languages have run-time optimisers that can generate fast code approaching the speed of static language compilers, often by using partial type inference.