Statements and Language Programming

Alex Burks

STATEMENTS AND LANGUAGE PROGRAMMING

STATEMENTS AND LANGUAGE PROGRAMMING

Alex Burks

BIBLIOTEX
Digital Library

Statements and Language Programming
by Alex Burks

Ebook ISBN: 9781984665225

Published by:
Bibliotex
Canada
Website: www.bibliotex.com

Contents

Chapter 1

Statement (Computer Science)

In computer programming, a **statement** is a syntactic unit of an imperative programming language that expresses some action to be carried out. A program written in such a language is formed by a sequence of one or more statements. A statement may have internal components (e.g., expressions).

Many programming languages (e.g. Ada, Algol 60, C, Java, Pascal) make a distinction between statements and definitions/declarations. A definition or declaration specifies the data on which a program is to operate, while a statement specifies the actions to be taken with that data.

Statements which cannot contain other statements are *simple*; those which can contain other statements are *compound*.

The appearance of a statement (and indeed a program) is determined by its syntax or grammar. The meaning of a statement is determined by its semantics.

Simple statements

Simple statements are complete in themselves; these include assignments, subroutine calls, and a few statements which may significantly affect the program flow of control (e.g. goto, return, stop/halt). In some languages, input and output, assertions, and exits are handled by special statements, while other languages use calls to predefined subroutines.

- assignment
- Fortran: `variable = expression`
- Pascal, Algol 60, Ada: `variable := expression;`
- C, C#, C++, PHP, Java: `variable = expression;`
- call
- Fortran: `CALL subroutine name(parameters)`
- C, C++, Java, PHP, Pascal, Ada: `subroutine name(parameters);`
- assertion
- C, C++, PHP: `assert(relational expression);`
- Java: `assert relational expression;`
- goto
- Fortran: `GOTO numbered-label`
- Algol 60: **goto**`label;`
- C, C++, PHP, Pascal: `gotolabel;`
- return
- Fortran: `RETURN value`
- C, C++, Java, PHP: `return value;`
- stop/halt/exit
- Fortran: `STOP number`
- C, C++: `exit(expression)`
- PHP: `exit number;`

Compound statements

Compound statements may contain (sequences of) statements, nestable to any reasonable depth, and generally involve tests to decide whether or not to obey or repeat these contained statements.

- Notation for the following examples:

- <statement> is any single statement (could be simple or compound).
- <sequence> is any sequence of zero or more <statements>
- Some programming languages provide a general way of grouping statements together, so that any single <statement> can be replaced by a group:
- Algol 60: `**begin**<sequence>**end**`
- Pascal: `begin <sequence> end`
- C, PHP, Java: `{ <sequence> }`
- Other programming languages have a different special terminator on each kind of compound statement, so that one or more statements are automatically treated as a group:
- Ada: `if test then <sequence> end if;`

Many compound statements are loop commands or choice commands. In theory only one of each of these types of commands is required. In practice there are various special cases which occur quite often; these may make a program easier to understand, may make programming easier, and can often be implemented much more efficiently. There are many subtleties not mentioned here; see the linked articles for details.

- count-controlled loop:
- Algol 60: **for** `index := 1` **step** `1` **until** `limit` **do**`<statement> ;`
- Pascal: `for index := 1 to limit do <statement> ;`
- C, Java: `for ( index = 1; index <= limit; index += 1) <statement> ;`
- Ada: `for index in 1..limit loop <sequence> end loop`
- Fortran 90:

- DO index = 1,limit
- <sequence>
- END DO
- condition-controlled loop with test at start of loop:
- Algol 60: `for index := expression while test do<statement> ;`
- Pascal: `while test do <statement> ;`
- C, Java: `while (test) <statement> ;`
- Ada: `while test loop <sequence> end loop`
- Fortran 90:
- DO WHILE (test)
- <sequence>
- END DO
- condition-controlled loop with test at end of loop:
- Pascal: `repeat <sequence> until test; { note reversed test}`
- C, Java: `do { <sequence> } while (test) ;`
- Ada: `loop <sequence> exit when test; end loop;`
- condition-controlled loop with test in the middle of the loop:
- C: `do { <sequence> if (test) break; <sequence> } while (true) ;`
- Ada: `loop <sequence> exit when test; <sequence> end loop;`
- if-statement simple situation:
- Algol 60:`if test then<unconditional statement> ;`
- Pascal:`if test then <statement> ;`
- C, Java: `if (test) <statement> ;`
- Ada: `if test then <sequence> end if;`
- Fortran 77+:
- IF (test) THEN
- <sequence>
- END IF

- if-statement two-way choice:
- Algol 60:`**if**` `test` `**then**<unconditional statement>**else**<statement> ;`
- Pascal:`if test then <statement> else <statement> ;`
- C, Java: `it (test) <statement> else <statement> ;`
- Ada: `if test then <sequence> else <sequence> end if;`
- Fortran 77+:
- IF (test) THEN
- <sequence>
- ELSE
- <sequence>
- END IF
- case/switch statement multi-way choice:
- Pascal: `case c of 'a': alert(); 'q': quit(); end;`
- Ada: `case c is when 'a' => alert(); when 'q' => quit(); end case;`
- C, Java: `switch (c) { case 'a': alert(); break; case 'q': quit(); break; }`
- Exception handling:
- Ada: `begin` *`protected code`* `except when` *`exception specification`* `=>`*`exception handler`*
- Java: `try {` *`protected code`* `} catch (`*`exception specification`*`) {` *`exception handler`* `} finally {` *`cleanup`* `}`
- Python: `try:` *`protected code`* `except` *`exception specification`*`:` *`exception handler`* `else:` *`no exceptions`* `finally:` *`cleanup`*

Syntax

Apart from assignments and subroutine calls, most languages start each statement with a special word (e.g. goto, if, while, etc.) as shown in the above examples. Various methods have been used to describe the form of statements in different languages; the more formal methods tend to be more precise:

- Algol 60 used Backus–Naur form (BNF) which set a new level for language grammar specification.
- Up until Fortran 77, the language was described in English prose with examples, From Fortran 90 onwards, the language was described using a variant of BNF.
- Cobol used a two-dimensional metalanguage.
- Pascal used both syntax diagrams and equivalent BNF.

BNF uses recursion to express repetition, so various extensions have been proposed to allow direct indication of repetition.

Statements and keywords

Some programming language grammars reserve keywords or mark them specially, and do not allow them to be used as identifiers. This often leads to grammars which are easier to parse, requiring less lookahead.

No distinguished keywords

Fortran and PL/1 do not have reserved keywords, allowing statements like:

- in PL/1:
- `IF IF = THEN THEN ...` (the second `IF` and the first `THEN` are variables).
- in Fortran:

- `IF (A) X = 10...` conditional statement (with other variants)
- `IF (A) = 2` assignment to a subscripted variable named `IF`
- As spaces were optional up to Fortran 95, a typo could completely change the meaning of a statement:
- `DO 10 I = 1,5` start of a loop with I running from 1 to 5
- `DO 10 I = 1.5` assignment of the value 1.5 to the variable `DO10I`

Flagged words

In Algol 60 and Algol 68, special tokens were distinguished explicitly: for publication, in boldface e.g. **`begin`**; for programming, with some special marking, e.g., a flag (`'begin`), quotation marks (`'begin'`), or underlined (`begin` on the Elliott 503). This is called "stropping".

Tokens that are part of the language syntax thus do not conflict with programmer-defined names.

Reserved keywords

Certain names are reserved as part of the programming language and can not be used as programmer-defined names. The majority of the most popular programming languages use reserved keywords. Early examples include FLOW-MATIC (1953) and COBOL (1959). Since 1970 other examples include Ada, C, C++, Java, and Pascal. The number of reserved words depends on the language: C has about 30 while COBOL has about 400.

Semantics

In programming language theory, **semantics** is the field concerned with the rigorous mathematical study of the meaning of programming languages. It does so by evaluating the meaning of syntactically valid strings defined by a specific programming language, showing the computation involved. In such a case that the evaluation would be of syntactically invalid strings, the result would be non-computation. Semantics describes the processes a computer follows when executing a program in that specific language. This can be shown by describing the relationship between the input and output of a program, or an explanation of how the program will be executed on a certain platform, hence creating a model of computation. The field of formal semantics encompasses all of the following:

- The definition of semantic models
- The relations between different semantic models
- The relations between different approaches to meaning
- The relation between computation and the underlying mathematical structures from fields such as logic, set theory, model theory, category theory, etc.

It has close links with other areas of computer science such as programming language design, type theory, compilers and interpreters, program verification and model checking.

Approaches

There are many approaches to formal semantics; these belong to three major classes:

- **Denotational semantics**, whereby each phrase in the language is interpreted as a *denotation*, i.e. a conceptual meaning that can be thought of abstractly. Such denotations are often mathematical objects inhabiting a mathematical space, but it is not a requirement that they should be so. As a practical necessity, denotations are described using some form of mathematical notation, which can in turn be formalized as a denotational metalanguage. For example, denotational semantics of functional languages often translate the language into domain theory. Denotational semantic descriptions can also serve as compositional translations from a programming language into the denotational metalanguage and used as a basis for designing compilers.
- **Operational semantics**, whereby the execution of the language is described directly (rather than by translation). Operational semantics loosely corresponds to interpretation, although again the "implementation language" of the interpreter is generally a mathematical formalism. Operational semantics may define an abstract machine (such as the SECD machine), and give meaning to phrases by describing the transitions they induce on states of the machine. Alternatively, as with the pure lambda

calculus, operational semantics can be defined via syntactic transformations on phrases of the language itself;

- **Axiomatic semantics**, whereby one gives meaning to phrases by describing the *axioms* that apply to them. Axiomatic semantics makes no distinction between a phrase's meaning and the logical formulas that describe it; its meaning *is* exactly what can be proven about it in some logic. The canonical example of axiomatic semantics is Hoare logic.

Apart from the choice between denotational, operational, or axiomatic approaches, most variations in formal semantic systems arise from the choice of supporting mathematical formalism.

Variations

Some variations of formal semantics include the following:

- **Action semantics** is an approach that tries to modularize denotational semantics, splitting the formalization process in two layers (macro and microsemantics) and predefining three semantic entities (actions, data and yielders) to simplify the specification;
- **Algebraic semantics** is a form of axiomatic semantics based on algebraic laws for describing and reasoning about program semantics in a formal manner;
- **Attribute grammars** define systems that systematically compute "metadata" (called *attributes*)

for the various cases of the language's syntax. Attribute grammars can be understood as a denotational semantics where the target language is simply the original language enriched with attribute annotations. Aside from formal semantics, attribute grammars have also been used for code generation in compilers, and to augment regular or context-free grammars with context-sensitive conditions;

- **Categorical (or "functorial") semantics** uses category theory as the core mathematical formalism. A categorical semantics is usually proven to correspond to some axiomatic semantics that gives a syntactic presentation of the categorical structures. Also, denotational semantics are often instances of a general categorical semantics,
- **Concurrency semantics** is a catch-all term for any formal semantics that describes concurrent computations. Historically important concurrent formalisms have included the actor model and process calculi;
- **Game semantics** uses a metaphor inspired by game theory.
- **Predicate transformer semantics**, developed by Edsger W. Dijkstra, describes the meaning of a program fragment as the function transforming a postcondition to the precondition needed to establish it.

Describing relationships

For a variety of reasons, one might wish to describe the relationships between different formal semantics. For example:

- To prove that a particular operational semantics for a language satisfies the logical formulas of an axiomatic semantics for that language. Such a proof demonstrates that it is "sound" to reason about a particular (operational) *interpretation strategy* using a particular (axiomatic) *proof system*.

- To prove that operational semantics over a high-level machine is related by a simulation with the semantics over a low-level machine, whereby the low-level abstract machine contains more primitive operations than the high-level abstract machine definition of a given language. Such a proof demonstrates that the low-level machine "faithfully implements" the high-level machine.

It is also possible to relate multiple semantics through abstractions via the theory of abstract interpretation.

History

Robert W. Floyd is credited with founding the field of programming language semantics in Floyd (1967).

Robert W. Floyd

Robert W Floyd (June 8, 1936 – September 25, 2001) was a computer scientist. His contributions include the design of the Floyd–Warshall algorithm (independently of Stephen Warshall), which efficiently finds all shortest paths in a graph, Floyd's cycle-finding algorithm for detecting cycles in a sequence, and his work on parsing. In one isolated paper he introduced the important concept of error diffusion for rendering images, also called Floyd–Steinberg dithering (though he distinguished dithering from diffusion). He pioneered in the field of program verification using logical assertions with the 1967 paper *Assigning Meanings to Programs*. This was a contribution to what later became Hoare logic. Floyd received the Turing Award in 1978.

Life

Born in New York City, Floyd finished high school at age 14. At the University of Chicago, he received a Bachelor of Arts (B.A.) in liberal arts in 1953 (when still only 17) and a second bachelor's degree in physics in 1958. Floyd was a college roommate of Carl Sagan.

Floyd became a staff member of the Armour Research Foundation (now IIT Research Institute) at Illinois Institute of Technology in the 1950s. Becoming a computer operator in the early 1960s, he began publishing many papers, including on compilers (particularly parsing). He was a pioneer of operator-precedence grammars, and is credited with initiating the field of programming language semantics in Floyd (1967). He was appointed an associate professor at Carnegie Mellon University by the time he was 27 and became a full professor at Stanford University six years later. He obtained this position without a Doctor of Philosophy (Ph.D.) degree.

He was a member of the International Federation for Information Processing (IFIP) IFIP Working Group 2.1 on Algorithmic Languages and Calculi, which specified, maintains, and supports the programming languages ALGOL 60 and ALGOL 68.

He was elected a Fellow of the American Academy of Arts and Sciences in 1974. He received the Turing Award in 1978 "for having a clear influence on methodologies for the creation of efficient and reliable software, and for helping to found the following important subfields of computer science: the theory of parsing, the semantics of programming languages, automatic program verification, automatic program synthesis, and analysis of algorithms".

Floyd worked closely with Donald Knuth, in particular as the major reviewer for Knuth's seminal book *The Art of Computer Programming*, and is the person most cited in that work. He was co-author, with Richard Beigel, of the textbook *The Language of Machines: an Introduction to Computability and Formal Languages*. Floyd supervised seven Ph.D. graduates.

Floyd married and divorced twice, first with Jana M. Mason and then computer scientist Christiane Floyd, and he had four children. In his last years he suffered from Pick's disease, a neurodegenerative disease, and thus retired early in 1994.

His hobbies included hiking, and he was an avid backgammon player:

We once were stuck at the Chicago O'Hare airport for hours, waiting for our flight to leave, owing to a snow storm. As we sat at our gate, Bob asked me, in a casual manner, "do you know how to play backgammon?" I answered I knew the rules, but why did he want to know? Bob said since we had several hours to wait perhaps we should play a few games, for small stakes of course. He then reached into his briefcase and removed a backgammon set.

My Dad taught me many things. One was to be wary of anyone who suggests a game of pool for money, and then opens a black case and starts to screw together a pool stick. I figured that this advice generalized to anyone who traveled with their own backgammon set. I told Bob that I was not going to play for money, no way. He pushed a bit, but finally said fine. He proceeded instead to give me a free lesson in the art and science of playing backgammon.

I was right to pass on playing him for money—at any stakes. The lesson was fun. I found out later that for years he had been working on learning the game. He took playing backgammon very seriously, studied the game and its mathematics, and was a near professional. I think it was more than a hobby. Like his research, Bob took what he did seriously, and it is completely consistent that he would be terrific at backgammon.

- —*Richard J. Lipton.*

Chapter 2

Programming Language

A **programming language** is a formal language comprising a set of strings that produce various kinds of machine code output. Programming languages are one kind of computer language, and are used in computer programming to implement algorithms.

Most programming languages consist of instructions for computers. There are programmable machines that use a set of specific instructions, rather than general programming languages. Since the early 1800s, programs have been used to direct the behavior of machines such as Jacquard looms, music boxes and player pianos. The programs for these machines (such as a player piano's scrolls) did not produce different behavior in response to different inputs or conditions.

Thousands of different programming languages have been created, and more are being created every year. Many programming languages are written in an imperative form (i.e., as a sequence of operations to perform) while other languages use the declarative form (i.e. the desired result is specified, not how to achieve it).

The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). 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 treated as a reference. Some languages have both, with the basic language defined by a standard and extensions taken from the dominant implementation being common.

Programming language theory is a subfield of computer science that deals with the design, implementation, analysis, characterization, and classification of programming languages.

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm. Some 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 involves 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 a 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-92 and Charity are examples of languages that are not Turing complete, yet are often called programming languages.

Markup languages like XML, HTML, or troff, which define structured data, are not usually 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 language entirely using XML syntax. 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. Similarly, 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.

History

Early developments

Very early computers, such as Colossus, were programmed without the help of a stored program, by modifying their circuitry or setting banks of physical controls.

Slightly later, programs could be written in machine language, where the programmer writes each instruction in a numeric form the hardware can execute directly. For example, the instruction to add the value in two memory locations might consist of 3 numbers: an "opcode" that selects the "add" operation, and two memory locations. The programs, in decimal or binary form, were read in from punched cards, paper tape, magnetic tape or toggled in on switches on the front panel of the computer. Machine languages were later termed *first-generation programming languages* (1GL).

The next step was the development of the so-called *second-generation programming languages* (2GL) or assembly languages, which were still closely tied to the instruction set architecture of the specific computer. These served to make the program much more human-readable and relieved the programmer of tedious and error-prone address calculations.

The first *high-level programming languages*, or *third-generation programming languages* (3GL), were written in the 1950s. An early 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.

John Mauchly's Short Code, proposed in 1949, was one of the first high-level languages ever developed for an electronic computer. Unlike machine code, Short Code statements represented mathematical expressions in understandable form. However, the program had to be translated into machine code every time it ran, making the process much slower than running the equivalent machine code.

At the University of Manchester, AlickGlennie developed Autocode in the early 1950s. As a programming language, it used a compiler to automatically convert the language into machine code. The first code and compiler was developed in 1952 for the Mark 1 computer at the University of Manchester and is considered to be the first compiled high-level programming language.

The second autocodewas developed for the Mark 1 by R. A. Brooker in 1954 and was called the "Mark 1 Autocode". Brooker also developed an autocode for the Ferranti Mercury in the 1950s in conjunction with the University of Manchester. The version for the EDSAC 2 was devised by D. F. Hartley of University of Cambridge Mathematical Laboratory in 1961. Known as EDSAC 2 Autocode, it was a straight development from Mercury Autocode adapted for local circumstances and was noted for its object code optimisation and source-language diagnostics which were advanced for the time. A contemporary but separate thread of development, Atlas Autocodewas developed for the University of Manchester Atlas 1 machine.

In 1954, FORTRAN was invented at IBM by John Backus. It was the first widely used high-level general purpose programming language to have a functional implementation, as opposed to just a design on paper. It is still a popular language for high-performance computing and is used for programs that benchmark and rank the world's fastest supercomputers.

Another early programming language was devised by Grace Hopper in the US, called FLOW-MATIC. It was developed for the UNIVAC I at Remington Rand during the period from 1955 until 1959. Hopper found that business data processing customers were uncomfortable with mathematical notation, and in early 1955, she and her team wrote a specification for an English programming language and implemented a prototype. The FLOW-MATIC compiler became publicly available in early 1958 and was substantially complete in 1959. FLOW-MATIC was a major influence in the design of COBOL, since only it and its direct descendant AIMACO were in actual use at the time.

Refinement

The increased use of high-level languages introduced a requirement for *low-level programming languages* or *system programming languages*. These languages, to varying degrees, provide facilities between assembly languages and high-level languages. They can be used to perform tasks that require direct access to hardware facilities but still provide higher-level control structures and error-checking.

The period from the 1960s to the late 1970s brought the development of the major language paradigms now in use:

- APL introduced *array programming* and influenced functional programming.
- ALGOL refined both *structured procedural programming* and the discipline of language specification; the "Revised Report on the Algorithmic Language ALGOL 60" became a model for how later language specifications were written.
- Lisp, implemented in 1958, was the first dynamically typed *functional programming* language.
- 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 for the Unix operating system 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 descendants, and most modern programming 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. EdsgerDijkstra, 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.

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 the 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 decades.

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, which 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 of many existing languages and paradigms (although their syntax was often 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.

Fourth-generation programming languages (4GL) are computer programming languages that aim to provide a higher level of abstraction of the internal computer hardware details than 3GLs. *Fifth-generation programming languages* (5GL) are programming languages based on solving problems using constraints given to the program, rather than using an algorithm written by a programmer.

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

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, this article discusses textual syntax.

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`, `()` and `(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 operations that are not semantically defined (the operation `*p >> 4` has no meaning for a value having a complex type and `p->im` is not defined because the value of `p` is the null pointer):

```
complex*p=NULL;
complexabs_p=sqrt(*p>>4+p->im);
```

If the type declaration on the first line were omitted, the program would trigger an error on undefined variable "p" during compilation. However, 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 an 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. For example, the data represented by `"this text between the quotes"` is a string, and in many programming languages dividing a number by a string has no meaning and will not be executed. The invalid operation may be detected when the program is compiled ("static" type checking) and will be rejected by the compiler with a compilation error message, or it may be detected while the program is running ("dynamic" type checking), resulting in a run-time exception. Many languages allow a function called an exception handler to handle this exception and, for example, always return "-1" as the result.

A special case of typed languages are the *single-typed* 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, generally sequences of bits of various lengths. High-level untyped languages include BCPL, Tcl, and some varieties of Forth.

In practice, while few languages are considered typed from the 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, trading type-safety for finer control over the program's execution (see casting).

Static versus dynamic typing

In *static typing*, all expressions have their types determined prior to when the program is executed, 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, C++, Java and C# all infer types in certain limited cases. Additionally, some programming languages allow for some types to be automatically converted to other types; for example, an int can be used where the program expects a float.

Dynamic typing, also called *latent typing*, determines the type-safety of operations at run time; in other words, types are associated with *run-time 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. Lisp, Smalltalk, Perl, Python, JavaScript, and Ruby are all examples of dynamically typed languages.

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 run time.

Strong typing prevents these program faults. 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.

It may seem odd to some professional programmers that C could be "weakly, statically typed". However, notice that the use of the generic pointer, the **void*** pointer, does allow for casting of pointers to other pointers without needing to do an explicit cast. This is extremely similar to somehow casting an array of bytes to any kind of datatype in C without using an explicit cast, such as `(int)` or `(char)`.

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.

The line between a language and its core library differs from language to language. In some cases, the language designers may treat the library as a separate entity from the language. However, a language's core library is often treated as part of the language by its users, and some language specifications even require that this library be made available in all implementations. 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 programming languages have been designed from scratch, altered to meet new needs, and combined with other languages. Many have 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.
- 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 programming has 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 an artifact that the language users and the implementors can use to agree upon whether a piece of source code is a valid program in that language, and if so what its behavior shall be.

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 write programs in that language and execute them 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 much 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.

Proprietary languages

Although most of the most commonly used programming languages have fully open specifications and implementations, many programming languages exist only as proprietary programming languages with the implementation available only from a single vendor, which may claim that such a proprietary language is their intellectual property. Proprietary programming languages are commonly domain specific languages or internal scripting languages for a single product; some proprietary languages are used only internally within a vendor, while others are available to external users. Some programming languages exist on the border between proprietary and open; for example, Oracle Corporation asserts proprietary rights to some aspects of the Java programming language, and Microsoft's C# programming language, which has open implementations of most parts of the system, also has Common Language Runtime (CLR) as a closed environment.

Many proprietary languages are widely used, in spite of their proprietary nature; examples include MATLAB, VBScript, and Wolfram Language. Some languages may make the transition from closed to open; for example, Erlang was originally an Ericsson's internal programming language.

Use

Thousands of different programming languages have been created, mainly in the computing field. Individual software projects commonly use five programming languages or more.

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. On the other hand, ideas about an algorithm can be communicated to humans without the precision required for execution by using pseudocode, which interleaves natural language with code written in a programming language.

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 can run its commands through an interpreter (such as a Unix shell or other command-line interface), without compiling, it is called a scripting language.

Measuring language usage

Determining which is the most widely used programming language is difficult since the definition of usage varies by context. One language may occupy the greater number of programmer hours, a different one has more lines of code, and a third may consume 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; Ada in aerospace, transportation, military, real-time and embedded applications; and C in embedded applications and operating systems. 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, stackify.com reported the ten most popular programming languages as (in descending order by overall popularity): Java, C, C++, Python, C#, JavaScript, VB .NET, R, PHP, and MATLAB.

Dialects, flavors and implementations

A **dialect** of a programming language or a data exchange language is a (relatively small) variation or extension of the language that does not change its intrinsic nature. With languages such as Scheme and Forth, standards may be considered insufficient, inadequate or illegitimate by implementors, so often they will deviate from the standard, making a new dialect. In other cases, a dialect is created for use in a domain-specific language, often a subset. In the Lisp world, most languages that use basic S-expression syntax and Lisp-like semantics are considered Lisp dialects, although they vary wildly, as do, say, Racket and Clojure. As it is common for one language to have several dialects, it can become quite difficult for an inexperienced programmer to find the right documentation. The BASIC programming language has many dialects. The explosion of Forth dialects led to the saying "If you've seen one Forth... you've seen *one* Forth."

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*, with general-purpose programming languages distinguished from domain-specific programming languages. 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 deliberately esoteric or not.

Chapter 3

Ada, ALGOL and APL

Ada (programming language)

Ada is a structured, statically typed, imperative, and object-oriented high-level programming language, extended from Pascal and other languages. It has built-in language support for *design by contract* (DbC), extremely strong typing, explicit concurrency, tasks, synchronous message passing, protected objects, and non-determinism. Ada improves code safety and maintainability by using the compiler to find errors in favor of runtime errors. Ada is an international technical standard, jointly defined by the International Organization for Standardization (ISO), and the International Electrotechnical Commission (IEC). As of 2020, the standard, called Ada 2012 informally, is ISO/IEC 8652:2012.

Ada was originally designed by a team led by French computer scientist Jean Ichbiah of CII Honeywell Bull under contract to the United States Department of Defense (DoD) from 1977 to 1983 to supersede over 450 programming languages used by the DoD at that time. Ada was named after Ada Lovelace (1815–1852), who has been credited as the first computer programmer.

Features

Ada was originally designed for embedded and real-time systems. The Ada 95 revision, designed by S. Tucker Taft of

Intermetrics between 1992 and 1995, improved support for systems, numerical, financial, and object-oriented programming (OOP).

Features of Ada include: strong typing, modular programming mechanisms (packages), run-time checking, parallel processing (tasks, synchronous message passing, protected objects, and nondeterministic select statements), exception handling, and generics. Ada 95 added support for object-oriented programming, including dynamic dispatch.

The syntax of Ada minimizes choices of ways to perform basic operations, and prefers English keywords (such as "or else" and "and then") to symbols (such as "||" and "&&"). Ada uses the basic arithmetical operators "+", "-", "*", and "/", but avoids using other symbols. Code blocks are delimited by words such as "declare", "begin", and "end", where the "end" (in most cases) is followed by the identifier of the block it closes (e.g., *if ... end if*, *loop ... end loop*). In the case of conditional blocks this avoids a *dangling else* that could pair with the wrong nested if-expression in other languages like C or Java.

Ada is designed for developing very large software systems. Ada packages can be compiled separately. Ada package specifications (the package interface) can also be compiled separately without the implementation to check for consistency. This makes it possible to detect problems early during the design phase, before implementation starts.

A large number of compile-time checks are supported to help avoid bugs that would not be detectable until run-time in some other languages or would require explicit checks to be added to the source code. For example, the syntax requires explicitly named closing of blocks to prevent errors due to mismatched end tokens. The adherence to strong typing allows detecting many common software errors (wrong parameters, range violations, invalid references, mismatched types, etc.) either during compile-time, or otherwise during run-time. As concurrency is part of the language specification, the compiler can in some cases detect potential deadlocks. Compilers also commonly check for misspelled identifiers, visibility of packages, redundant declarations, etc. and can provide warnings and useful suggestions on how to fix the error.

Ada also supports run-time checks to protect against access to unallocated memory, buffer overflow errors, range violations, off-by-one errors, array access errors, and other detectable bugs. These checks can be disabled in the interest of runtime efficiency, but can often be compiled efficiently. It also includes facilities to help program verification. For these reasons, Ada is widely used in critical systems, where any anomaly might lead to very serious consequences, e.g., accidental death, injury or severe financial loss. Examples of systems where Ada is used include avionics, air traffic control, railways, banking, military and space technology.

Ada's dynamic memory management is high-level and type-safe. Ada has no generic or untyped pointers; nor does it implicitly declare any pointer type. Instead, all dynamic memory allocation and deallocation must occur via explicitly declared *access types*. Each access type has an associated *storage pool* that handles the low-level details of memory management; the programmer can either use the default storage pool or define new ones (this is particularly relevant for Non-Uniform Memory Access). It is even possible to declare several different access types that all designate the same type but use different storage pools. Also, the language provides for *accessibility checks*, both at compile time and at run time, that ensures that an *access value* cannot outlive the type of the object it points to.

Though the semantics of the language allow automatic garbage collection of inaccessible objects, most implementations do not support it by default, as it would cause unpredictable behaviour in real-time systems. Ada does support a limited form of region-based memory management; also, creative use of storage pools can provide for a limited form of automatic garbage collection, since destroying a storage pool also destroys all the objects in the pool.

A double-dash ("--"), resembling an em dash, denotes comment text. Comments stop at end of line, to prevent unclosed comments from accidentally voiding whole sections of source code. Disabling a whole block of code now requires the prefixing of each line (or column) individually with "--". While clearly denoting disabled code with a column of repeated "--" down the page this renders the experimental dis/re-enablement of large blocks a more drawn out process.

The semicolon (";") is a statement terminator, and the null or no-operation statement is `null;`. A single `;` without a statement to terminate is not allowed.

Unlike most ISO standards, the Ada language definition (known as the *Ada Reference Manual* or *ARM*, or sometimes the *Language Reference Manual* or *LRM*) is free content. Thus, it is a common reference for Ada programmers, not only programmers implementing Ada compilers. Apart from the reference manual, there is also an extensive rationale document which explains the language design and the use of various language constructs. This document is also widely used by programmers. When the language was revised, a new rationale document was written.

One notable free software tool that is used by many Ada programmers to aid them in writing Ada source code is the GNAT Programming Studio, part of the GNU Compiler Collection.

History

In the 1970s the US Department of Defense (DoD) became concerned by the number of different programming languages being used for its embedded computer system projects, many of which were obsolete or hardware-dependent, and none of which supported safe modular programming. In 1975, a working group, the High Order Language Working Group (HOLWG), was formed with the intent to reduce this number by finding or creating a programming language generally suitable for the department's and the UK Ministry of Defence's requirements. After many iterations beginning with an original Straw man proposal the eventual programming language was named Ada. The total number of high-level programming languages in use for such projects fell from over 450 in 1983 to 37 by 1996.

The HOLWG working group crafted the Steelman language requirements, a series of documents stating the requirements they felt a programming language should satisfy. Many existing languages were formally reviewed, but the team concluded in 1977 that no existing language met the specifications.

Requests for proposals for a new programming language were issued and four contractors were hired to develop their proposals under the names of Red (Intermetrics led by Benjamin Brosgol), Green (CII Honeywell Bull, led by Jean Ichbiah), Blue (SofTech, led by John Goodenough) and Yellow (SRI International, led by Jay Spitzen). In April 1978, after public scrutiny, the Red and Green proposals passed to the next phase. In May 1979, the Green proposal, designed by Jean Ichbiah at CII Honeywell Bull, was chosen and given the name Ada—after Augusta Ada, Countess of Lovelace. This proposal was influenced by the language LIS that Ichbiah and his group had developed in the 1970s. The preliminary Ada reference manual was published in ACM SIGPLAN Notices in June 1979. The Military Standard reference manual was approved on December 10, 1980 (Ada Lovelace's birthday), and given the number MIL-STD-1815 in honor of Ada Lovelace's birth year. In 1981, C. A. R. Hoare took advantage of his Turing Award speech to criticize Ada for being overly complex and hence unreliable, but subsequently seemed to recant in the foreword he wrote for an Ada textbook.

Ada attracted much attention from the programming community as a whole during its early days. Its backers and others predicted that it might become a dominant language for general purpose programming and not only defense-related work. Ichbiah publicly stated that within ten years, only two programming languages would remain: Ada and Lisp. Early Ada compilers struggled to implement the large, complex language, and both compile-time and run-time performance tended to be slow and tools primitive. Compiler vendors expended most of their efforts in passing the massive, language-conformance-testing, government-required "ACVC" validation suite that was required in another novel feature of the Ada language effort. The Jargon File, a dictionary of computer hacker slang originating in 1975–1983, notes in an entry on Ada that "it is precisely what one might expect given that kind of endorsement by fiat; designed by committee...difficult to use, and overall a disastrous, multi-billion-dollar boondoggle...Ada Lovelace...would almost certainly blanch at the use her name has been latterly put to; the kindest thing that has been said about it is that there is probably a good small language screaming to get out from inside its vast, elephantine bulk."

The first validated Ada implementation was the NYU Ada/Ed translator, certified on April 11, 1983. NYU Ada/Ed is implemented in the high-level set language SETL. Several commercial companies began offering Ada compilers and associated development tools, including Alsys, TeleSoft, DDC-I, Advanced Computer Techniques, Tartan Laboratories, TLD Systems, and Verdix.

In 1991, the US Department of Defense began to require the use of Ada (the *Ada mandate*) for all software, though exceptions to this rule were often granted. The Department of Defense Ada mandate was effectively removed in 1997, as the DoD began to embrace commercial off-the-shelf (COTS) technology. Similar requirements existed in other NATO countries: Ada was required for NATO systems involving command and control and other functions, and Ada was the mandated or preferred language for defense-related applications in countries such as Sweden, Germany, and Canada.

By the late 1980s and early 1990s, Ada compilers had improved in performance, but there were still barriers to fully exploiting Ada's abilities, including a tasking model that was different from what most real-time programmers were used to.

Because of Ada's safety-critical support features, it is now used not only for military applications, but also in commercial projects where a software bug can have severe consequences, e.g., avionics and air traffic control, commercial rockets such as the Ariane 4 and 5, satellites and other space systems, railway transport and banking. For example, the Airplane Information Management System, the fly-by-wire system software in the Boeing 777, was written in Ada. Developed by Honeywell Air Transport Systems in collaboration with consultants from DDC-I, it became arguably the best-known of any Ada project, civilian or military. The Canadian Automated Air Traffic System was written in 1 million lines of Ada (SLOC count). It featured advanced distributed processing, a distributed Ada database, and object-oriented design. Ada is also used in other air traffic systems, e.g., the UK's next-generation Interim Future Area Control Tools Support (iFACTS) air traffic control system is designed and implemented using SPARK Ada. It is also used in the French TVM in-cab signalling system on the TGV high-speed rail system, and the metro suburban trains in Paris, London, Hong Kong and New York City.

Standardization

The language became an ANSI standard in 1983 (ANSI/MIL-STD 1815A), and after translation in French and without any further changes in English became an ISO standard in 1987

(ISO-8652:1987). This version of the language is commonly known as Ada 83, from the date of its adoption by ANSI, but is sometimes referred to also as Ada 87, from the date of its adoption by ISO.

Ada 95, the joint ISO/ANSI standard (ISO-8652:1995) was published in February 1995, making Ada 95 the first ISO standard object-oriented programming language. To help with the standard revision and future acceptance, the US Air Force funded the development of the GNAT Compiler. Presently, the GNAT Compiler is part of the GNU Compiler Collection.

Work has continued on improving and updating the technical content of the Ada language. A Technical Corrigendum to Ada 95 was published in October 2001, and a major Amendment, ISO/IEC 8652:1995/Amd 1:2007 was published on March 9, 2007. At the Ada-Europe 2012 conference in Stockholm, the Ada Resource Association (ARA) and Ada-Europe announced the completion of the design of the latest version of the Ada language and the submission of the reference manual to the International Organization for Standardization (ISO) for approval. ISO/IEC 8652:2012 was published in December 2012.

Other related standards include ISO 8651-3:1988 *Information processing systems—Computer graphics—Graphical Kernel System (GKS) language bindings—Part 3: Ada*.

Language constructs

Ada is an ALGOL-like programming language featuring control structures with reserved words such as *if*, *then*, *else*, *while*, *for*, and so on. However, Ada also has many data structuring facilities and other abstractions which were not included in the original ALGOL 60, such as type definitions, records, pointers, enumerations. Such constructs were in part inherited from or inspired by Pascal.

"Hello, world!" in Ada

A common example of a language's syntax is the Hello world program: (hello.adb)

```
withAda.Text_IO;useAda.Text_IO;
procedureHellois
begin
Put_Line("Hello, world!");
endHello;
```

This program can be compiled by using the freely available open source compiler GNAT, by executing

```
gnatmakehello.adb
```

Data types

Ada's type system is not based on a set of predefined primitive types but allows users to declare their own types. This declaration in turn is not based on the internal representation of the type but on describing the goal which should be achieved. This allows the compiler to determine a suitable memory size for the type, and to check for violations of the type definition at compile time and run time (i.e., range violations, buffer overruns, type consistency, etc.). Ada supports numerical types defined by a range, modulo types, aggregate types (records and arrays), and enumeration types. Access types define a reference to an instance of a specified type; untyped pointers are not permitted. Special types provided by the language are task types and protected types.

For example, a date might be represented as:

```
typeDay_typeisrange1..31;
typeMonth_typeisrange1..12;
typeYear_typeisrange1800..2100;
typeHoursismod24;
typeWeekdayis(Monday,Tuesday,Wednesday,Thursday,Friday,Saturday,Sunday);

typeDateis
record
Day:Day_type;
Month:Month_type;
Year:Year_type;
end record;
```

Types can be refined by declaring subtypes:

```
subtypeWorking_HoursisHoursrange0..12;-- at most 12 Hours to work a day
subtypeWorking_DayisWeekdayrangeMonday..Friday;-- Days to work

Work_Load:constantarray(Working_Day)ofWorking_Hours-- implicit type declaration
:=(Friday=>6,Monday=>4,others=>10);-- lookup table for working hours with initialization
```

Types can have modifiers such as *limited, abstract, private* etc. Private types can only be accessed and limited types can only be modified or copied within the scope of the package that defines them. Ada 95 adds further features for object-oriented extension of types.

Control structures

Ada is a structured programming language, meaning that the flow of control is structured into standard statements. All standard constructs and deep-level early exit are supported, so the use of the also supported "go to" commands is seldom needed.

```
-- while a is not equal to b, loop.
whilea/=bloop
```

```
Ada.Text_IO.Put_Line("Waiting");
endloop;

ifa>bthen
Ada.Text_IO.Put_Line("Condition met");
else
Ada.Text_IO.Put_Line("Condition not met");
endif;

foriin1..10loop
Ada.Text_IO.Put("Iteration: ");
Ada.Text_IO.Put(i);
Ada.Text_IO.Put_Line;
endloop;

loop
a:=a+1;
exitwhena=10;
endloop;

caseiis
when0=>Ada.Text_IO.Put("zero");
when1=>Ada.Text_IO.Put("one");
when2=>Ada.Text_IO.Put("two");
-- case statements have to cover all possible cases:
whenothers=>Ada.Text_IO.Put("none of the above");
endcase;

foraWeekdayinWeekday'Rangeloop-- loop over an enumeration
Put_Line(Weekday'Image(aWeekday));-- output string representation of an enumeration
ifaWeekdayinWorking_Daythen-- check of a subtype of an enumeration
Put_Line(" to work for "&
Working_Hours'Image(Work_Load(aWeekday)));-- access into a lookup table
endif;
endloop;
```

Packages, procedures and functions

Among the parts of an Ada program are packages, procedures and functions.

Example: Package specification (example.ads)

```
packageExampleis
typeNumberisrange1..11;
procedurePrint_and_Increment(j: inoutNumber);
endExample;
```

Package body (example.adb)

```
withAda.Text_IO;
packagebodyExampleis
```

```
i:Number:=Number'First;

procedurePrint_and_Increment(j: inoutNumber)is

functionNext(k: inNumber)returnNumberis
begin
returnk+1;
endNext;

begin
Ada.Text_IO.Put_Line("The total is: "&Number'Image(j));
j:=Next(j);
endPrint_and_Increment;

-- package initialization executed when the package is elaborated
begin
whilei<Number'Lastloop
Print_and_Increment(i);
endloop;
endExample;
```

This program can be compiled, e.g., by using the freely available open-source compiler GNAT, by executing

```
gnatmake -z example.adb
```

Packages, procedures and functions can nest to any depth, and each can also be the logical outermost block.

Each package, procedure or function can have its own declarations of constants, types, variables, and other procedures, functions and packages, which can be declared in any order.

Concurrency

Ada has language support for task-based concurrency. The fundamental concurrent unit in Ada is a *task*, which is a built-in limited type. Tasks are specified in two parts – the task declaration defines the task interface (similar to a type declaration), the task body specifies the implementation of the task. Depending on the implementation, Ada tasks are either mapped to operating system threads or processes, or are scheduled internally by the Ada runtime.

Tasks can have entries for synchronisation (a form of synchronous message passing). Task entries are declared in the task specification. Each task entry can have one or more *accept* statements within the task body. If the control flow of the task reaches an accept statement, the task is blocked until the corresponding entry is called by another task (similarly, a calling task is blocked until the called task reaches the corresponding accept statement). Task entries can have parameters similar to procedures, allowing tasks to synchronously exchange data. In conjunction with *select* statements it is possible to define *guards* on accept statements (similar to Dijkstra's guarded commands).

Ada also offers *protected objects* for mutual exclusion. Protected objects are a monitor-like construct, but use guards instead of conditional variables for signaling (similar to conditional critical regions). Protected objects combine the data encapsulation and safe mutual exclusion from monitors, and entry guards from conditional critical regions. The main advantage over classical monitors is that conditional variables are not required for signaling, avoiding potential deadlocks due to incorrect locking semantics. Like tasks, the protected object is a built-in limited type, and it also has a declaration part and a body.

A protected object consists of encapsulated private data (which can only be accessed from within the protected object), and procedures, functions and entries which are guaranteed to be mutually exclusive (with the only exception of functions, which are required to be side effect free and can therefore run concurrently with other functions). A task calling a protected object is blocked if another task is currently executing inside the same protected object, and released when this other task leaves the protected object. Blocked tasks are queued on the protected object ordered by time of arrival.

Protected object entries are similar to procedures, but additionally have *guards*. If a guard evaluates to false, a calling task is blocked and added to the queue of that entry; now another task can be admitted to the protected object, as no task is currently executing inside the protected object. Guards are re-evaluated whenever a task leaves the protected object, as this is the only time when the evaluation of guards can have changed.

Calls to entries can be *requeued* to other entries with the same signature. A task that is requeued is blocked and added to the queue of the target entry; this means that the protected object is released and allows admission of another task.

The *select* statement in Ada can be used to implement non-blocking entry calls and accepts, non-deterministic selection of entries (also with guards), time-outs and aborts.

The following example illustrates some concepts of concurrent programming in Ada.

```
withAda.Text_IO;useAda.Text_IO;

procedureTrafficis

typeAirplane_IDisrange1..10;-- 10 airplanes

tasktypeAirplane(ID: Airplane_ID);-- task representing airplanes, with ID as initialisation
parameter
typeAirplane_AccessisaccessAirplane;-- reference type to Airplane
```

```
protectedtypeRunwayis-- the shared runway (protected to allow concurrent access)
entryAssign_Aircraft(ID: Airplane_ID);-- all entries are guaranteed mutually exclusive
entryCleared_Runway(ID: Airplane_ID);
entryWait_For_Clear;
private
Clear:Boolean:=True;-- protected private data - generally more than only a flag...
endRunway;
typeRunway_AccessisaccessallRunway;

-- the air traffic controller task takes requests for takeoff and landing
tasktypeController(My_Runway: Runway_Access)is
-- task entries for synchronous message passing
entryRequest_Takeoff(ID: inAirplane_ID;Takeoff: outRunway_Access);
entryRequest_Approach(ID: inAirplane_ID;Approach: outRunway_Access);
endController;

--  allocation of instances
Runway1:aliasedRunway;-- instantiate a runway
Controller1:Controller(Runway1'Access);-- and a controller to manage it

------ the implementations of the above types ------
protectedbodyRunwayis
entryAssign_Aircraft(ID: Airplane_ID)
whenClearis-- the entry guard - calling tasks are blocked until the condition is true
begin
Clear:=False;
Put_Line(Airplane_ID'Image(ID)&" on runway ");
end;

entryCleared_Runway(ID: Airplane_ID)
whennotClearis
begin
Clear:=True;
Put_Line(Airplane_ID'Image(ID)&" cleared runway ");
end;

entryWait_For_Clear
whenClearis
begin
null;-- no need to do anything here - a task can only enter if "Clear" is true
end;
endRunway;

taskbodyControlleris
begin
loop
My_Runway.Wait_For_Clear;-- wait until runway is available (blocking call)
select-- wait for two types of requests (whichever is runnable first)
whenRequest_Approach'count=0=>-- guard statement - only accept if there are no tasks
queuing on Request_Approach
acceptRequest_Takeoff(ID:inAirplane_ID;Takeoff:outRunway_Access)
do-- start of synchronized part
My_Runway.Assign_Aircraft(ID);-- reserve runway (potentially blocking call if protected
object busy or entry guard false)
Takeoff:=My_Runway;-- assign "out" parameter value to tell airplane which runway
endRequest_Takeoff;-- end of the synchronised part
or
acceptRequest_Approach(ID:inAirplane_ID;Approach:outRunway_Access)do
My_Runway.Assign_Aircraft(ID);
```

```
Approach:=My_Runway;
endRequest_Approach;
or-- terminate if no tasks left who could call
terminate;
endselect;
endloop;
end;

taskbodyAirplaneis
Rwy:Runway_Access;
begin
Controller1.Request_Takeoff(ID,Rwy);-- This call blocks until Controller task accepts and
completes the accept block
Put_Line(Airplane_ID'Image(ID)&"  taking off...");
delay2.0;
Rwy.Cleared_Runway(ID);-- call will not block as "Clear" in Rwy is now false and no other
tasks should be inside protected object
delay5.0;-- fly around a bit...
loop
select-- try to request a runway
Controller1.Request_Approach(ID,Rwy);-- this is a blocking call - will run on controller
reaching accept block and return on completion
exit;-- if call returned we're clear for landing - leave select block and proceed...
or
delay3.0;-- timeout - if no answer in 3 seconds, do something else (everything in following
block)
Put_Line(Airplane_ID'Image(ID)&"   in holding pattern");-- simply print a message
endselect;
endloop;
delay4.0;-- do landing approach...
Put_Line(Airplane_ID'Image(ID)&"            touched down!");
Rwy.Cleared_Runway(ID);-- notify runway that we're done here.
end;

New_Airplane:Airplane_Access;

begin
forIinAirplane_ID'Rangeloop-- create a few airplane tasks
New_Airplane:=newAirplane(I);-- will start running directly after creation
delay4.0;
endloop;
endTraffic;
```

Pragmas

A pragma is a compiler directive that conveys information to the compiler to allow specific manipulating of compiled output. Certain pragmas are built into the language, while others are implementation-specific.

Examples of common usage of compiler pragmas would be to disable certain features, such as run-time type checking or array subscript boundary checking, or to instruct the compiler to insert object code instead of a function call (as C/C++ does with inline functions).

Generics

Ada has had generics since it was first designed in 1977–1980. The standard library uses generics to provide many services. Ada 2005 adds a comprehensive generic container library to the standard library, which was inspired by C++'s standard template library.

A *generic unit* is a package or a subprogram that takes one or more *generic formal parameters*.

A *generic formal parameter* is a value, a variable, a constant, a type, a subprogram, or even an instance of another, designated, generic unit. For generic formal types, the syntax distinguishes between discrete, floating-point, fixed-point, access (pointer) types, etc. Some formal parameters can have default values.

- To *instantiate* a generic unit, the programmer passes *actual* parameters for each formal. The generic instance then behaves just like any other unit. It is possible to instantiate generic units at run-time, for example inside a loop.

ALGOL

ALGOL (/□æl□□l,-□ɔ : l/; short for "**Algorithmic Language**") is a family of imperative computer programming languages originally developed in 1958. ALGOL heavily influenced many other languages and was the standard method for algorithm description used by the Association for Computing Machinery (ACM) in textbooks and academic sources for more than thirty years.

In the sense that the syntax of most modern languages is "Algol-like", it was arguably the most influential of the four high-level programming languages among which it was roughly contemporary: FORTRAN, Lisp, and COBOL. It was designed to avoid some of the perceived problems with FORTRAN and eventually gave rise to many other programming languages, including PL/I, Simula, BCPL, B, Pascal, and C.

ALGOL introduced code blocks and the `begin`...`end` pairs for delimiting them. It was also the first language implementing nested function definitions with lexical scope. Moreover, it was the first programming language which gave detailed attention to formal language definition and through the *Algol 60 Report* introduced Backus–Naur form, a principal formal grammar notation for language design.

There were three major specifications, named after the years they were first published:

- ALGOL 58 – originally proposed to be called*IAL*, for *International Algebraic Language*.
- ALGOL 60 – first implemented as *X1 ALGOL 60* in mid-1960. Revised 1963.
- ALGOL 68 – introduced new elements including flexible arrays, slices, parallelism, operator identification. Revised 1973.

ALGOL 68 is substantially different from ALGOL 60 and was not well received, so in general "Algol" means ALGOL 60 and its dialects.

Important implementations

The International Algebraic Language (IAL), renamed ALGOL 58, was highly influential and is generally considered the ancestor of most modern programming languages (the so-called Algol-like languages).

ALGOL object code was a simple, compact stack-based instruction set architecture commonly used in teaching compiler construction and other high order languages.

History

ALGOL was developed jointly by a committee of European and American computer scientists in a meeting in 1958 at the Swiss Federal Institute of Technology in Zurich (cf. ALGOL 58). It specified three different syntaxes: a reference syntax, a publication syntax, and an implementation syntax. The different syntaxes permitted it to use different keyword names and conventions for decimal points (commas vs periods) for different languages.

ALGOL was used mostly by research computer scientists in the United States and in Europe. Its use in commercial applications was hindered by the absence of standard input/output facilities in its description and the lack of interest in the language by large computer vendors other than

Burroughs Corporation. ALGOL 60 did however become the standard for the publication of algorithms and had a profound effect on future language development.

John Backus developed the *Backus normal form* method of describing programming languages specifically for ALGOL 58. It was revised and expanded by Peter Naur for ALGOL 60, and at Donald Knuth's suggestion renamed Backus–Naur form.

Peter Naur: "As editor of the ALGOL Bulletin I was drawn into the international discussions of the language and was selected to be member of the European language design group in November 1959. In this capacity I was the editor of the ALGOL 60 report, produced as the result of the ALGOL 60 meeting in Paris in January 1960."

The following people attended the meeting in Paris (from 1 to 16 January):

- Friedrich L. Bauer, Peter Naur, Heinz Rutishauser, Klaus Samelson, Bernard Vauquois, Adriaan van Wijngaarden, and Michael Woodger (from Europe)
- John W. Backus, Julien Green, Charles Katz, John McCarthy, Alan J. Perlis, and Joseph Henry Wegstein (from the USA).

Alan Perlis gave a vivid description of the meeting: "The meetings were exhausting, interminable, and exhilarating. One became aggravated when one's good ideas were discarded along with the bad ones of others. Nevertheless, diligence persisted during the entire period. The chemistry of the 13 was excellent."

ALGOL 60 inspired many languages that followed it. Tony Hoareremarked: "Here is a language so far ahead of its time that it was not only an improvement on its predecessors but also on nearly all its successors." The Scheme programming language, a variant of Lisp that adopted the block structure and lexical scope of ALGOL, also adopted the wording "Revised Report on the Algorithmic Language Scheme" for its standards documents in homage to ALGOL.

ALGOL and programming language research

As Peter Landin noted, ALGOL was the first language to combine seamlessly imperative effects with the (call-by-name) lambda calculus. Perhaps the most elegant formulation of the language is due to John C. Reynolds, and it best exhibits its syntactic and semantic purity. Reynolds's idealized ALGOL also made a convincing methodologic argument regarding the suitability of local effects in the context of call-by-name languages, in contrast with the global effects used by call-by-value languages such as ML. The conceptual integrity of the language made it one of the main objects of semantic research, along with Programming Computable Functions (PCF) and ML.

Properties

ALGOL 60 as officially defined had no I/O facilities; implementations defined their own in ways that were rarely compatible with each other. In contrast, ALGOL 68 offered an extensive library of *transput* (input/output) facilities.

ALGOL 60 allowed for two evaluation strategies for parameter passing: the common call-by-value, and call-by-name. Call-by-name has certain effects in contrast to call-by-reference. For example, without specifying the parameters as *value* or *reference*, it is impossible to develop a procedure that will swap the values of two parameters if the actual parameters that are passed in are an integer variable and an array that is indexed by that same integer variable. Think of passing a pointer to swap(i,

A[i]) in to a function. Now that every time swap is referenced, it is reevaluated. Say i := 1 and A[i] := 2, so every time swap is referenced it will return the other combination of the values ([1,2], [2,1], [1,2] and so on). A similar situation occurs with a random function passed as actual argument.

Call-by-name is known by many compiler designers for the interesting "thunks" that are used to implement it. Donald Knuth devised the "man or boy test" to separate compilers that correctly implemented "recursion and non-local references." This test contains an example of call-by-name.

ALGOL 68 was defined using a two-level grammar formalism invented by Adriaan van Wijngaarden and which bears his name.

Van Wijngaarden grammars use a context-free grammar to generate an infinite set of productions that will recognize a particular ALGOL 68 program; notably, they are able to express the kind of requirements that in many other programming language standards are labelled "semantics" and have to be expressed in ambiguity-prone natural language prose, and then implemented in compilers as *ad hoc* code attached to the formal language parser.

Examples and portability issues

Code sample comparisons

ALGOL 60

(The way the bold text has to be written depends on the implementation, e.g. 'INTEGER'—quotation marks included—for **integer**. This is known as stropping.)

```
procedureAbsmax(a) Size:(n, m) Result:(y) Subscripts:(i, k);
value n, m; array a; integer n, m, i, k; real y;
comment The absolute greatest element of the matrix a, of size n by m
is transferred to y, and the subscripts of this element to i and k;
begin
integer p, q;
y := 0; i := k := 1;
for p := 1 step 1 until n do
for q := 1 step 1 until m do
if abs(a[p, q]) > y then
beginy := abs(a[p, q]);
i := p; k := q
end
endAbsmax
```

Here is an example of how to produce a table using Elliott 803 ALGOL.

```
 FLOATING POINT ALGOL TEST'
 BEGIN REAL A,B,C,D'

 READ D'

 FOR A:= 0.0 STEP D UNTIL 6.3 DO
 BEGIN
   PRINT PUNCH(3),££L??'
B := SIN(A)'
C := COS(A)'
   PRINT PUNCH(3),SAMELINE,ALIGNED(1,6),A,B,C'
 END'
 END'
```

PUNCH(3) sends output to the teleprinter rather than the tape punch.

SAMELINE suppresses the carriage return + line feed normally printed between arguments.

ALIGNED(1,6) controls the format of the output with 1 digit before and 6 after the decimal point.

ALGOL 68

The following code samples are ALGOL 68 versions of the above ALGOL 60 code samples.

ALGOL 68 implementations used ALGOL 60's approaches to stropping. In ALGOL 68's case tokens with the **bold** typeface are reserved words, types (**mode**s) or operators.

```
proc abs max = ([,]real a, refreal y, refinti, k)real:
comment The absolute greatest element of the matrix a, of size ⌈a by 2⌈a
is transferred to y, and the subscripts of this element to i and k; comment
begin
real y := 0; i := ⌊a; k := 2⌊a;
for p from⌊a to⌈a do
for q from 2⌊a to 2⌈a do
ifabs a[p, q] > y then
y := abs a[p, q];
i := p; k := q
fi
od
od;
y
end # abs max #
```

Note: lower (⌊) and upper (⌈) bounds of an array, and array slicing, are directly available to the programmer.

```
floating point algol68 test:
(
reala,b,c,d;

  # printf - sends output to the filestand out. #
  # printf($p$); – selects a new page #
printf(($pg$,"Enter d:"));
read(d);

for step from 0 while a:=step*d; a <= 2*pi do
printf($l$);  # $l$ - selects a new line. #
b := sin(a);
c := cos(a);
printf(($z-d.6d$,a,b,c))  # formats output with 1 digit before and 6 after the decimal point. #
od
)
```

Timeline: Hello world

The variations and lack of portability of the programs from one implementation to another is easily demonstrated by the classic hello world program.

ALGOL 58 (IAL)

ALGOL 58 had no I/O facilities.

ALGOL 60 family

Since ALGOL 60 had no I/O facilities, there is no portable hello world program in ALGOL. The next three examples are in Burroughs Extended Algol. The first two direct output at the interactive terminal they are run on. The first uses a character array, similar to C. The language allows the array identifier to be used as a pointer to the array, and hence in a REPLACE statement.

```
BEGIN
FILEF(KIND=REMOTE);
EBCDICARRAYE[0:11];
REPLACEEBY"HELLO WORLD!";
WRITE(F,*,E);
END.
```

A simpler program using an inline format:

```
BEGIN
FILEF(KIND=REMOTE);
WRITE(F,<"HELLO WORLD!">);
END.
```

An even simpler program using the Display statement. Note that its output would end up at the system console ('SPO'):

```
BEGINDISPLAY("HELLO WORLD!")END.
```

An alternative example, using Elliott Algol I/O is as follows. Elliott Algol used different characters for "open-string-quote" and "close-string-quote":

```
programHiFolks;
begin
print 'Hello world';
end;
```

Here is a version for the Elliott 803 Algol (A104) The standard Elliott 803 used 5 hole paper tape and thus only had upper case. The code lacked any quote characters so £ (UK Pound Sign) was used for open quote and ? (Question Mark) for close quote. Special sequences were placed in double quotes (e.g. ££L?? produced a new line on the teleprinter).

```
 HIFOLKS'
 BEGIN
   PRINT £HELLO WORLD£L??'
 END'
```

The ICT 1900 series Algol I/O version allowed input from paper tape or punched card. Paper tape 'full' mode allowed lower case. Output was to a line printer. The open and close quote characters were represented using '(' and ')' and spaces by %.

```
 'BEGIN'
   WRITE TEXT('('HELLO%WORLD')');
 'END'
```

ALGOL 68

- Main article: ALGOL 68

ALGOL 68 code was published with reserved words typically in lowercase, but bolded or underlined.

```
begin
printf(($gl$,"Hello, world!"))
end
```

In the language of the "Algol 68 Report" the input/output facilities were collectively called the "Transput".

Timeline of ALGOL special characters

The ALGOLs were conceived at a time when character sets were diverse and evolving rapidly; also, the ALGOLs were defined so that only *uppercase* letters were required.

1960: IFIP – The Algol 60 language and report included several mathematical symbols which are available on modern computers and operating systems, but, unfortunately, were unsupported on most computing systems at the time. For instance: ×, ÷, ≤, ≥, ≠, ¬, ∨, ∧, ⊂, ≡, ␣ and •.

1961 September: ASCII – The ASCII character set, then in an early stage of development, had the \ (Back slash) character added to it in order to support ALGOL's boolean operators /\ and \/.

1962: ALCOR – This character set included the unusual "+" runic cross character for multiplication and the "•" Decimal Exponent Symbol for floating point notation.

1964: GOST – The 1964 Soviet standard GOST 10859 allowed the encoding of 4-bit, 5-bit, 6-bit and 7-bit characters in ALGOL.

1968: The "Algol 68 Report" – used extant ALGOL characters, and further adopted →, ↓, ↑, □, ⌊, ⌈, ⎩, ⎧, ○, ⊥, and ¢ characters which can be found on the IBM 2741 keyboard with *typeball* (or *golf ball*) print heads inserted (such as the APL golf ball).

These became available in the mid-1960s while ALGOL 68 was being drafted.

The report was translated into Russian, German, French, and Bulgarian, and allowed programming in languages with larger character sets, e.g., Cyrillic alphabet of the Soviet BESM-4. All ALGOL's characters are also part of the Unicode standard and most of them are available in several popular fonts.

2009 October: Unicode – The □ (Decimal Exponent Symbol) for floating point notation was added to Unicode 5.2 for backward compatibility with historic Buran programme ALGOL software.

APL (programming language)

APL (named after the book *A Programming Language*) is a programming language developed in the 1960s by Kenneth E. Iverson. Its central datatype is the multidimensional array. It uses a large range of special graphic symbols to represent most functions and operators, leading to very concise code.

It has been an important influence on the development of concept modeling, spreadsheets, functional programming, and computer math packages. It has also inspired several other programming languages.

History

Mathematical notation

A mathematical notation for manipulating arrays was developed by Kenneth E. Iverson, starting in 1957 at Harvard University. In 1960, he began work for IBM where he developed this notation with Adin Falkoff and published it in his book *A Programming Language* in 1962. The preface states its premise:

Applied mathematics is largely concerned with the design and analysis of explicit procedures for calculating the exact or approximate values of various functions. Such explicit procedures are called algorithms or *programs*. Because an effective notation for the description of programs exhibits considerable syntactic structure, it is called a *programming language*.

This notation was used inside IBM for short research reports on computer systems, such as the Burroughs B5000 and its stack mechanism when stack machines versus register machines were being evaluated by IBM for upcoming computers.

Iverson also used his notation in a draft of the chapter *A Programming Language*, written for a book he was writing with Fred Brooks, *Automatic Data Processing*, which would be published in 1963.

In 1979, Iverson received the Turing Award for his work on APL.

Development into a computer programming language

As early as 1962, the first attempt to use the notation to describe a complete computer system happened after Falkoff discussed with William C. Carter his work to standardize the instruction set for the machines that later became the IBM System/360 family.

In 1963, Herbert Hellerman, working at the IBM Systems Research Institute, implemented a part of the notation on an IBM 1620 computer, and it was used by students in a special high school course on calculating transcendental functions by series summation. Students tested their code in Hellerman's lab. This implementation of a part of the notation was called Personalized Array Translator (PAT).

In 1963, Falkoff, Iverson, and Edward H. Sussenguth Jr., all working at IBM, used the notation for a formal description of the IBM System/360 series machine architecture and functionality, which resulted in a paper published in *IBM Systems Journal* in 1964. After this was published, the team turned their attention to an implementation of the notation on a computer system. One of the motivations for this focus of implementation was the interest of John L. Lawrence who had new duties with Science Research Associates, an educational company bought by IBM in 1964. Lawrence asked Iverson and his group to help use the language as a tool to develop and use computers in education.

After Lawrence M. Breed and Philip S. Abrams of Stanford University joined the team at IBM Research, they continued their prior work on an implementation programmed in

FORTRAN IV for a part of the notation which had been done for the IBM 7090 computer running on the IBSYS operating system. This work was finished in late 1965 and later named IVSYS (for Iverson system). The basis of this implementation was described in detail by Abrams in a Stanford University Technical Report, "An Interpreter for Iverson Notation" in 1966, the academic aspect of this was formally supervised by Niklaus Wirth. Like Hellerman's PAT system earlier, this implementation did not include the APL character set but used special English reserved words for functions and operators. The system was later adapted for a time-sharing system and, by November 1966, it had been reprogrammed for the IBM System/360 Model 50 computer running in a time sharing mode and was used internally at IBM.

Hardware

A key development in the ability to use APL effectively, before the wide use of cathode ray tube (CRT) terminals, was the development of a special IBM Selectric typewriter interchangeable typing element with all the special APL characters on it. This was used on paper printing terminal workstations using the Selectric typewriter and typing element mechanism, such as the IBM 1050 and IBM 2741 terminal. Keycaps could be placed over the normal keys to show which APL characters would be entered and typed when that key was struck. For the first time, a programmer could type in and see proper APL characters as used in Iverson's notation and not be forced to use awkward English keyword representations of them. Falkoff and Iverson had the special APL Selectric typing elements, 987 and 988, designed in late 1964, although no APL computer system was available to use them. Iverson cited

Falkoff as the inspiration for the idea of using an IBM Selectric typing element for the APL character set.

Many APL symbols, even with the APL characters on the Selectric typing element, still had to be typed in by over-striking two extant element characters. An example is the *grade up* character, which had to be made from a *delta* (shift-H) and a *Sheffer stroke* (shift-M). This was necessary because the APL character set was much larger than the 88 characters allowed on the typing element, even when letters were restricted to upper-case (capitals).

Commercial availability

The first APL interactive login and creation of an APL workspace was in 1966 by Larry Breed using an IBM 1050 terminal at the IBM Mohansic Labs near Thomas J. Watson Research Center, the home of APL, in Yorktown Heights, New York.

IBM was chiefly responsible for introducing APL to the marketplace. APL was first available in 1967 for the IBM 1130 as *APL\1130*. It would run in as little as 8k 16-bit words of memory, and used a dedicated 1 megabyte hard disk.

APL gained its foothold on mainframe timesharing systems from the late 1960s through the early 1980s, in part because it would support multiple users on lower-specification systems that had no dynamic address translation hardware. Additional improvements in performance for selected IBM System/370 mainframe systems included the *APL Assist Microcode* in which some support for APL execution was included in the processor's firmware, as distinct from being implemented entirely by higher-level software. Somewhat later, as suitably performing hardware was finally growing available in the mid- to late-1980s, many users migrated their applications to the personal computer environment.

Early IBM APL interpreters for IBM 360 and IBM 370 hardware implemented their own multi-user management instead of relying on the host services, thus they were their own timesharing systems. First introduced in 1966, the *APL\360* system was a multi-user interpreter. The ability to programmatically communicate with the operating system for information and setting interpreter system variables was done through special privileged "I-beam" functions, using both monadic and dyadic operations.

In 1973, IBM released *APL.SV*, which was a continuation of the same product, but which offered shared variables as a means to access facilities outside of the APL system, such as operating system files. In the mid-1970s, the IBM mainframe interpreter was even adapted for use on the IBM 5100 desktop computer, which had a small CRT and an APL keyboard, when most other small computers of the time only offered BASIC. In the 1980s, the *VSAPL* program product enjoyed wide use with Conversational Monitor System (CMS), Time Sharing Option (TSO), VSPC, MUSIC/SP, and CICS users.

In 1973–1974, Patrick E. Hagerty directed the implementation of the University of Maryland APL interpreter for the 1100 line of the Sperry UNIVAC 1100/2200 series mainframe computers. At the time, Sperry had nothing. In 1974, student Alan Stebbenswas assigned the task of implementing an internal function. Xerox APL was available from June 1975 for Xerox

560 and Sigma 6, 7, and 9 mainframes running CP-V and for Honeywell CP-6.

In the 1960s and 1970s, several timesharing firms arose that sold APL services using modified versions of the IBM APL\360 interpreter. In North America, the better-known ones were I. P. Sharp Associates, Scientific Time Sharing Corporation (STSC), Time Sharing Resources (TSR), and The Computer Company (TCC). CompuServe also entered the market in 1978 with an APL Interpreter based on a modified version of Digital Equipment Corp and Carnegie Mellon's, which ran on DEC's KI and KL 36-bit machines. CompuServe's APL was available both to its commercial market and the consumer information service. With the advent first of less expensive mainframes such as the IBM 4300, and later the personal computer, by the mid-1980s, the timesharing industry was all but gone.

Sharp APL was available from I. P. Sharp Associates, first as a timesharing service in the 1960s, and later as a program product starting around 1979. *Sharp APL* was an advanced APL implementation with many language extensions, such as *packages* (the ability to put one or more objects into a single variable), file system, nested arrays, and shared variables.

APL interpreters were available from other mainframe and mini-computer manufacturers also, notably Burroughs, Control Data Corporation (CDC), Data General, Digital Equipment Corporation (DEC), Harris, Hewlett-Packard (HP), Siemens AG, Xerox, and others.

Garth Foster of Syracuse University sponsored regular meetings of the APL implementers' community at Syracuse's Minnowbrook Conference Center in Blue Mountain Lake, New

York. In later years, Eugene McDonnell organized similar meetings at the Asilomar Conference Grounds near Monterey, California, and at Pajaro Dunes near Watsonville, California. The SIGAPL special interest group of the Association for Computing Machinery continues to support the APL community.

Microcomputers

On microcomputers, which became available from the mid 1970s onwards, BASIC became the dominant programming language. Nevertheless, some microcomputers provided APL instead - the first being the Intel 8008-based MCM/70 which was released in 1974 and which was primarily used in education. Another machine of this time was the VideoBrain Family Computer, released in 1977, which was supplied with its dialect of APL called APL/S.

The Commodore SuperPET, introduced in 1981, included an APL interpreter developed by the University of Waterloo.

In 1976, Bill Gates claimed in his Open Letter to Hobbyists that Microsoft Corporation was implementing APL for the Intel 8080 and Motorola 6800 but had "very little incentive to make [it] available to hobbyists" because of software piracy. It was never released.

APL2

Starting in the early 1980s, IBM APL development, under the leadership of Jim Brown, implemented a new version of the APL language that contained as its primary enhancement the concept of *nested arrays*, where an array can contain other arrays, and new language features which facilitated integrating nested arrays into program workflow. Ken Iverson, no longer in control of the development of the APL language, left IBM and joined I. P. Sharp Associates, where one of his major contributions was directing the evolution of Sharp APL to be more in accord with his vision.

As other vendors were busy developing APL interpreters for new hardware, notably Unix-based microcomputers, APL2 was almost always the standard chosen for new APL interpreter developments. Even today, most APL vendors or their users cite APL2 compatibility, as a selling point for those products.

APL2 for IBM mainframe computers is still available. IBM cites its use for problem solving, system design, prototyping, engineering and scientific computations, expert systems, for teaching mathematics and other subjects, visualization and database access and was first available for CMS and TSO in 1984. The APL2 Workstation edition (Windows, OS/2, AIX, Linux, and Solaris) followed much later in the early 1990s.

Modern implementations

Various implementations of APL by APLX, Dyalog, et al., include extensions for object-oriented programming, support for .NET Framework, XML-array conversion primitives, graphing, operating system interfaces, and lambda calculus expressions.

Derivative languages

APL has formed the basis of, or influenced, the following languages:

- A and A+, an alternative APL, the latter with graphical extensions.
- FP, a functional programming language.
- Ivy, an interpreter for an APL-like language developed by Rob Pike, and which uses ASCII as input.
- J, which was also designed by Iverson, and which uses ASCII with digraphs instead of special symbols.
- K, a proprietary variant of APL developed by Arthur Whitney.
- LYaPAS, a Soviet extension to APL.
- MATLAB, a numerical computation tool.
- Nial, a high-level array programming language with a functional programming notation.
- Polymorphic Programming Language, an interactive, extensible language with a similar base language.
- S, a statistical programming language (usually now seen in the open-source version known as R).
- Speakeasy, a numerical computing interactive environment.
- Wolfram Language, the programming language of Mathematica.

Language characteristics

Character set

APL has been both criticized and praised for its choice of a unique, non-standard character set. Some who learn it become ardent adherents, suggesting that there is some weight behind Iverson's idea that the notation used does make a difference.

In the 1960s and 1970s, few terminal devices and even display monitors could reproduce the APL character set. The most popular ones employed the IBM Selectric print mechanism used with a special APL type element. One of the early APL line terminals (line-mode operation only, *not* full screen) was the Texas Instruments TI Model 745 (circa 1977) with the full APL character set which featured half and full duplex telecommunications modes, for interacting with an APL time-sharing service or remote mainframe to run a remote computer job, called an RJE.

Over time, with the universal use of high-quality graphic displays, printing devices and Unicode support, the APL character font problem has largely been eliminated. However, entering APL characters requires the use of input method editors, keyboard mappings, virtual/on-screen APL symbol sets, or easy-reference printed keyboard cards which can frustrate beginners accustomed to other programming languages. With beginners who have no prior experience with other programming languages, a study involving high school students found that typing and using APL characters did not hinder the students in any measurable way.

In defense of APL use, APL requires less coding to type in, and keyboard mappings become memorized over time. Also, special APL keyboards are manufactured and in use today, as are freely available downloadable fonts for operating systems such as Microsoft Windows. The reported productivity gains assume that one will spend enough time working in APL to make it worthwhile to memorize the symbols, their semantics, and keyboard mappings, not to mention a substantial number of idioms for common tasks.

Design

Unlike traditionally structured programming languages, APL code is typically structured as chains of monadic or dyadic functions, and operators acting on arrays. APL has many nonstandard *primitives* (functions and operators) that are indicated by a single symbol or a combination of a few symbols. All primitives are defined to have the same precedence, and always associate to the right. Thus, APL is *read* or best understood from right-to-left.

Early APL implementations (circa 1970 or so) had no programming loop-flow control structures, such as `do` or`while` loops, and `if-then-else` constructs. Instead, they used array operations, and use of structured programming constructs was often not necessary, since an operation could be performed on a full array in one statement. For example, the `iota` function (ι) can replace for-loop iteration: ιN when applied to a scalar positive integer yields a one-dimensional array (vector), 1 2 3 ... N. More recent implementations of APL generally include comprehensive control structures, so that data structure and program control flow can be clearly and cleanly separated.

The APL environment is called a *workspace*. In a workspace the user can define programs and data, i.e., the data values exist also outside the programs, and the user can also manipulate the data without having to define a program. In the examples below, the APL interpreter first types six spaces before awaiting the user's input. Its own output starts in column one.

APL uses a set of non-ASCII symbols, which are an extension of traditional arithmetic and algebraic notation. Having single character names for single instruction, multiple data (SIMD) vector functions is one way that APL enables compact formulation of algorithms for data transformation such as computing Conway's Game of Life in one line of code. In nearly all versions of APL, it is theoretically possible to express any computable function in one expression, that is, in one line of code.

Because of the unusual character set, many programmers use special keyboards with APL keytops to write APL code. Although there are various ways to write APL code using only ASCII characters, in practice it is almost never done. (This may be thought to support Iverson's thesis about notation as a tool of thought.) Most if not all modern implementations use standard keyboard layouts, with special mappings or input method editors to access non-ASCII characters. Historically, the APL font has been distinctive, with uppercase italic alphabetic characters and upright numerals and symbols. Most vendors continue to display the APL character set in a custom font.

Advocates of APL claim that the examples of so-called *write-only code* (badly written and almost incomprehensible code) are almost invariably examples of poor programming practice or novice mistakes, which can occur in any language. Advocates also claim that they are far more productive with APL than with more conventional computer languages, and that working software can be implemented in far less time and with far fewer programmers than using other technology.

They also may claim that because it is compact and terse, APL lends itself well to larger-scale software development and complexity, because the number of lines of code can be reduced greatly. Many APL advocates and practitioners also view standard programming languages such as COBOL and Java as being comparatively tedious. APL is often found where time-to-market is important, such as with trading systems.

Terminology

APL makes a clear distinction between *functions* and *operators*. Functions take arrays (variables or constants or expressions) as arguments, and return arrays as results. Operators (similar to higher-order functions) take functions or arrays as arguments, and derive related functions. For example, the *sum* function is derived by applying the *reduction* operator to the *addition* function. Applying the same reduction operator to the *maximum* function (which returns the larger of two numbers) derives a function which returns the largest of a group (vector) of numbers. In the J language, Iverson substituted the terms *verb* for *function* and *adverb* or *conjunction* for *operator*.

APL also identifies those features built into the language, and represented by a symbol, or a fixed combination of symbols, as *primitives*. Most primitives are either functions or operators. Coding APL is largely a process of writing non-primitive functions and (in some versions of APL) operators. However a few primitives are considered to be neither functions nor operators, most noticeably assignment.

Syntax

APL has explicit representations of functions, operators, and syntax, thus providing a basis for the clear and explicit statement of extended facilities in the language, and tools to experiment on them.

Examples

Hello, World

This displays "Hello, world":

```
'Hello, world'
```

A design theme in APL is to define default actions in some cases that would produce syntax errors in most other programming languages.

The 'Hello, world' string constant above displays, because display is the default action on any expression for which no action is specified explicitly (e.g. assignment, function parameter).

Exponentiation

Another example of this theme is that exponentiation in APL is written as "`2⋆3`", which indicates raising 2 to the power 3 (this would be written as "`2^3`" in some other languages and "`2⋆⋆3`" in FORTRAN and Python): many languages use * to signify multiplication as in 2*3 but APL uses `2×3` for that. However, if no base is specified (as with the statement "`⋆3`" in APL, or "`^3`" in other languages), in most other programming languages one would have a syntax error. APL however assumes the missing base to be the natural logarithm constant e (2.71828....), and so interpreting "`⋆3`" as "`2.71828⋆3`".

Simple statistics

Suppose that `X` is an array of numbers. Then `(+/X)÷⍴X` gives its average. Reading *right-to-left*, `⍴X` gives the number of elements in X, and since `÷` is a dyadic operator, the term to its left is required as well. It is in parenthesis since otherwise X would be taken (so that the summation would be of `X÷⍴X`, of each element of X divided by the number of elements in X), and `+/X` adds all the elements of X. Building on this, `((+/((X-(+/X)÷⍴X)*2))÷⍴X)*0.5` calculates the standard deviation. Further, since assignment is an operator, it can appear within an expression, so

```
SD←((+/((X-AV←(T←+/X)÷⍴X)*2))÷⍴X)*0.5
```

would place suitable values into T, AV and SD. Naturally, one would make this expression into a function for repeated use rather than retyping it each time.

Pick 6 **lottery numbers**

This following immediate-mode expression generates a typical set of *Pick 6* lottery numbers: six pseudo-random integers ranging from 1 to 40, *guaranteed non-repeating*, and displays them sorted in ascending order:

```
x[⍋x←6?40]
```

The above does a lot, concisely; although it seems complex to a new APLer. It combines the following APL *functions* (also called *primitives* and *glyphs*):

- The first to be executed (APL executes from rightmost to leftmost) is dyadic function `?`

(named`deal` when dyadic) that returns a vector consisting of a select number (left argument: 6 in this case) of random integers ranging from 1 to a specified maximum (right argument: 40 in this case), which, if said maximum ≥ vector length, is guaranteed to be non-repeating; thus, generate/create 6 random integers ranging from 1-40.

- This vector is then *assigned* (←) to the variable `x`, because it is needed later.
- This vector is then *sorted* in ascending order by a monadic ▯ function, which has as its right argument everything to the right of it up to the next unbalanced *close-bracket* or close-parenthesis. The result of ▯ is the indices that will put its argument into ascending order.
- Then the output of ▯is used to index the variable `x`, which we saved earlier for this purpose, thereby selecting its items in *ascending* sequence.

Since there is no function to the left of the left-most x to tell APL what to do with the result, it simply outputs it to the display (on a single line, separated by spaces) without needing any explicit instruction to do that.

`?`also has a monadic equivalent called `roll`, which simply returns one random integer between 1 and its sole operand [to the right of it], inclusive. Thus, a role-playing game program might use the expression `?20` to roll a twenty-sided die.

Prime numbers

The following expression finds all prime numbers from 1 to R.

In both time and space, the calculation complexity is (in Big O notation).

```
(~R∊R∘.×R)/R←1↓⍰R
```

Executed from right to left, this means:

- *Iota*⍰ creates a vector containing integers from `1` to `R` (if `R= 6` at the start of the program, `⍰R` is `1 2 3 4 5 6`)
- *Drop* first element of this vector (`↓` function), i.e., `1`. So `1↓⍰R` is `2 3 4 5 6`
- *Set*`R` to the new vector (`←`, *assignment* primitive), i.e., `2 3 4 5 6`
- The `/`*replicate* operator is dyadic (binary) and the interpreter first evaluates its left argument (fully in parentheses):
- Generate *outer product* of `R` multiplied by `R`, i.e., a matrix that is the *multiplication table* of R by R (`∘.×` operator), i.e.,
- Build a vector the same length as `R` with `1` in each place where the corresponding number in `R` is in the outer product matrix (`∊`, *set inclusion* or *element of* or *Epsilon* operator), i.e., `0 0 1 0 1`
- Logically negate (*not*) values in the vector (change zeros to ones and ones to zeros) (`∼`, logical *not* or *Tilde* operator), i.e., `1 1 0 1 0`
- Select the items in `R` for which the corresponding element is `1` (`/`*replicate* operator), i.e., `2 3 5`

(Note, this assumes the APL origin is 1, i.e., indices start with 1. APL can be set to use 0 as the origin, so that `⍳6` is `0 1 2 3 4 5`, which is convenient for some calculations.)

Sorting

The following expression sorts a word list stored in matrix X according to word length:

```
X[□X+.≠' ';]
```

Game of Life

The following function "life", written in Dyalog APL, takes a boolean matrix and calculates the new generation according to Conway's Game of Life. It demonstrates the power of APL to implement a complex algorithm in very little code, but it is also very hard to follow unless □ one has advanced knowledge of APL.

```
life←{↑1□∨.∧34=+/,¯101∘.⊖¯101∘. ⊂□}
```

HTML tags removal

In the following example, also Dyalog, the first line assigns some HTML code to a variable `txt` and then uses an APL expression to remove all the HTML tags (explanation):

```
txt←'<html><body><p>This is <em>emphasized</em> text.</p></body></html>'
{□/□~{□∨≠\□}□∊'<>'}txt
Thisisemphasizedtext.
```

Use

APL is used for many purposes including financial and insurance applications, artificial intelligence, neural networks and robotics. It has been argued that APL is a calculation tool and not a programming language; its symbolic nature and array capabilities have made it popular with domain experts and data scientists who do not have or require the skills of a computer programmer.

APL is well suited to image manipulation and computer animation, where graphic transformations can be encoded as matrix multiplications. One of the first commercial computer graphics houses, Digital Effects, produced an APL graphics product named *Visions*, which was used to create television commercials and animation for the 1982 film *Tron*. Latterly, the Stormwind boating simulator uses APL to implement its core logic, its interfacing to the rendering pipeline middleware and a major part of its physics engine.

Today, APL remains in use in a wide range of commercial and scientific applications, for example investment management, asset management, health care, and DNA profiling, and by hobbyists.

Notable implementations

APL\360

The first implementation of APL using recognizable APL symbols was APL\360 which ran on the IBM System/360, and was completed in November 1966 though at that time remained in use only within IBM. In 1973 its implementors, Larry Breed, Dick Lathwell and Roger Moore, were awarded the Grace Murray Hopper Award from the Association for Computing Machinery (ACM). It was given "for their work in the design and implementation of APL\360, setting new standards in simplicity, efficiency, reliability and response time for interactive systems."

In 1975, the IBM 5100 microcomputer offered APL\360 as one of two built-in ROM-based interpreted languages for the computer, complete with a keyboard and display that supported all the special symbols used in the language.

Significant developments to APL\360 included CMS/APL, which made use of the virtual storage capabilities of CMS and APLSV, which introduced shared variables, system variables and system functions. It was subsequently ported to the IBM System/370 and VSPC platforms until its final release in 1983, after which it was replaced by APL2.

APL\1130

In 1968, APL\1130 became the first publicly available APL system, created by IBM for the IBM 1130. It became the most popular IBM Type-III Library software that IBM released.

APL*Plus and Sharp APL

APL*Plus and Sharp APL are versions of APL\360 with added business-oriented extensions such as data formatting and facilities to store APL arrays in external files. They were jointly developed by two companies, employing various members of the original IBM APL\360 development team.

The two companies were I. P. Sharp Associates (IPSA), an APL\360 services company formed in 1964 by Ian Sharp, Roger Moore and others, and STSC, a time-sharing and consulting service company formed in 1969 by Lawrence Breed and others. Together the two developed APL*Plus and thereafter continued to work together but develop APL separately as APL*Plus and Sharp APL. STSC ported APL*Plus to many platforms with versions being made for the VAX 11, PC and UNIX, whereas IPSA took a different approach to the arrival of the Personal Computer and made Sharp APL available on this platform using additional PC-XT/360 hardware. In 1993, Soliton Incorporated was formed to support Sharp APL and it developed Sharp APL into SAX (Sharp APL for Unix). As of 2018, APL*Plus continues as APL2000 APL+Win.

In 1985, Ian Sharp, and Dan Dyer of STSC, jointly received the Kenneth E. Iverson Award for Outstanding Contribution to APL.

APL2

APL2 was a significant re-implementation of APL by IBM which was developed from 1971 and first released in 1984. It provides many additions to the language, of which the most notable is nested (non-rectangular) array support. As of 2018 it is available for mainframe computers running z/OS or z/VM and workstations running AIX, Linux, Sun Solaris, and Microsoft Windows.

The entire APL2 Products and Services Team was awarded the Iverson Award in 2007.

APLGOL

In 1972, APLGOL was released as an experimental version of APL that added structured programming language constructs to the language framework. New statements were added for interstatement control, conditional statement execution, and statement structuring, as well as statements to clarify the intent of the algorithm. It was implemented for Hewlett-Packard in 1977.

Dyalog APL

Dyalog APL was first released by British company Dyalog Ltd. in 1983 and, as of 2018, is available for AIX, Linux (including on the Raspberry Pi), macOS and Microsoft Windows platforms. It is based on APL2, with extensions to support object-oriented programming and functional programming. Licences are free for personal/non-commercial use.

In 1995, two of the development team - John Scholes and Peter Donnelly - were awarded the Iverson Award for their work on the interpreter. Gitte Christensen and Morten Kromberg were joint recipients of the Iverson Award in 2016.

NARS2000

NARS2000 is an open-source APL interpreter written by Bob Smith, a prominent APL developer and implementor from STSC in the 1970s and 1980s. NARS2000 contains advanced features and new datatypes and runs natively on Microsoft Windows, and other platforms under Wine.

APLX

APLX is a cross-platform dialect of APL, based on APL2 and with several extensions, which was first released by British company MicroAPL in 2002. Although no longer in development or on commercial sale it is now available free of charge from Dyalog.

GNU APL

GNU APL is a free implementation of Extended APL as specified in ISO/IEC 13751:2001 and is thus an implementation of APL2. It runs on Linux (including on the Raspberry Pi), macOS, several BSD dialects, and on Windows (either using Cygwin for full support of all its system functions or as a native 64-bit Windows binary with some of its system functions missing). GNU APL uses Unicode internally and can be scripted. It was written by Jürgen Sauermann.

Richard Stallman, founder of the GNU Project, was an early adopter of APL, using it to write a text editor as a high school student in the summer of 1969.

Interpretation and compilation of APL

APL is traditionally an interpreted language, having language characteristics such as weak variable typing not well suited to compilation. However, with arrays as its core data structure it provides opportunities for performance gains through parallelism, parallel computing, massively parallel applications, and very-large-scale integration (VLSI), and from the outset APL has been regarded as a high-performance language - for example, it was noted for the speed with which it could perform complicated matrix operations "because it operates on arrays and performs operations like matrix inversion internally". Nevertheless, APL is rarely purely interpreted and compilation or partial compilation techniques that are, or have been, used include the following:

Idiom recognition

Most APL interpreters support idiom recognition and evaluate common idioms as single operations. For example, by evaluating the idiom `BV/⍰⍰A` as a single operation (where `BV` is a Boolean vector and `A` is an array), the creation of two intermediate arrays is avoided.

Optimised bytecode

Weak typing in APL means that a name may reference an array (of any datatype), a function or an operator. In general, the interpreter cannot know in advance which form it will be and must therefore perform analysis, syntax checking etc. at run-time.

However, in certain circumstances, it is possible to deduce in advance what type a name is expected to reference and then generate bytecode which can be executed with reduced run-time overhead. This bytecode can also be optimised using compilation techniques such as constant folding or common subexpression elimination.

The interpreter will execute the bytecode when present and when any assumptions which have been made are met. Dyalog APL includes support for optimised bytecode.

Compilation

Compilation of APL has been the subject of research and experiment since the language first became available; the first compiler is considered to be the Burroughs APL-700 which was released around 1971.

In order to be able to compile APL, language limitations have to be imposed. APEX is a research APL compiler which was written by Robert Bernecky and is available under the GNU Public License.

The STSC APL Compiler is a hybrid of a bytecode optimiser and a compiler - it enables compilation of functions to machine code provided that its sub-functions and globals are declared, but the interpreter is still used as a runtime library and to execute functions which do not meet the compilation requirements.

Standards

APL has been standardized by the American National Standards Institute (ANSI) working group X3J10 and International Organization for Standardization (ISO) and International Electrotechnical Commission (IEC), ISO/IEC Joint Technical Committee 1 Subcommittee 22 Working Group 3.

The Core APL language is specified in ISO 8485:1989, and the Extended APL language is specified in ISO/IEC 13751:2001.

Chapter 4

Types of Statements

Press release

A **press release** is an official statement delivered to members of the news media for the purpose of providing information, creating an official statement, or making an announcement directed for public release. Press releases are also considered a primary source, meaning they are original informants for information. A press release is traditionally composed of nine structural elements, including a headline, dateline, introduction, body, and other components. Press releases are typically delivered to news media electronically, ready to use, and often subject to "do not use before" time, known as an news embargo.

A special example of a press release is a **communiqué** (/kə□mjuː n□ke□/), which is a brief report or statement released by a public agency. A communiqué is typically issued after a high-level meeting of international leaders.

Using press release material can benefit media corporations because they help decrease costs and improve the amount of material a media firm can output in a certain amount of time. Due to the material being pre-packaged, press releases save journalists time, not only in writing a story, but also the time and money it would have taken to capture the news firsthand.

Although using a press release can thus save a news outlet time and money, it constrains the format and style of its content. In addition, press releases are favorable towards the organization that commissioned them, framing the topic according to its preferred criteria. In the digital age, consumers want to get their information instantly, bringing about pressure on the news media to output as much material as possible. This may cause news media companies to heavily rely on press releases to create stories.

Elements

Any information deliberately sent to a reporter or media source is considered a press release. This information is released by the act of being sent to the media. Public relations professionals often follow a standard professional format for press releases. Additional communication methods that journalists employ include pitch letters and media advisories. Generally, a press release body consists of four to five paragraphs with a word limit ranging from 400 to 500. Press release length can range from 300 to 800 words.

Common structural elements include:

- **Letterhead or Logo**
- **Media Contact Information** – name, phone number, email address, mailing address, or other contact information for the public relation (PR) or other media relations contact person.
- **Headline** – used to grab the attention of journalists and briefly summarize the news in one to six words.
- **Dek** – a sub-headline that describes the headline in more detail.

- **Dateline** – contains the release date and usually the originating city of the press release. If the date listed is after the date that the information was actually sent to the media, then the sender is requesting a news embargo.
- **Introduction** – first paragraph in a press release, that generally gives basic answers to the questions of who, what, when, where and why.
- **Body** – further explanation, statistics, background, or other details relevant to the news.
- **Boilerplate** – generally a short "about" section, providing independent background on the issuing company, organization, or individual.
- **Close** – in North America, traditionally the symbol "-30-" appears after the boilerplate or body and before the media contact information, indicating to media that the release has ended. A more modern equivalent has been the "###" symbol. In other countries, other means of indicating the end of the release may be used, such as the text "ends".

As the Internet has assumed growing prominence in the news cycle, press release writing styles have evolved. Editors of online newsletters, for instance, often lack the staff to convert traditional press release prose into the print-ready copy.

Distribution models

In the traditional distribution model, the business, political campaign, or other entity releasing information to the media hires a publicity agency to write and distribute written information to the newswires. The newswire then scatters the information as it is received or as investigated by a journalist. Thus, resulting the information or announcement becoming public knowledge.

An alternative model is the *self-published press release*. In this approach, press releases are either sent directly to local newspapers or to free and paid distribution services. The distribution service then provides the content, as-is, to their media outlets for publication which is usually communicated via online. This approach is often used by political institutions, for example. Another instance would be, Constitutional Courts in Europe, U.S. Supreme Court, and the U.S. State Supreme Courts issue press releases about their own decisions and the news media use these self-published releases for their reporting.

Video

Some public relations firms send out video news releases (VNRs) which are pre-taped video programs or clips that can be aired intact by TV stations.

Video news releases may include interviews of movie-stars. These interviews, which have been taped on a set, are located at the movie studio and decorated with the movie's logo.

Video news releases can be in the form of full-blown productions as well. This costs tens of thousands or even hundreds of thousands of dollars to be produced. Video news releases can also be in the format of TV news, or even produced specifically for the web.

Some broadcast news outlets have discouraged the use of video news releases because of citing a poor public perception. It could also be viewed as a desire to increase their credibility.

Furthermore, VNRs can be turned into podcasts and then posted onto newswires. A story can also be kept running longer by simply engaging "community websites". "Community websites" are monitored and commented on by many journalists and feature writers.

Embargoes

If a press release is distributed before the information is intended to be released to the public it is considered embargoed. An embargo requests that news organizations not report the story until a specified date or time. Unless the journalist has signed a legally binding non-disclosure agreement agreeing to honor the embargo in advance, the journalist has no legal obligation to withhold the information. However, violating the embargo risks damaging their relationship with the issuing organization and their reputation as a writer or journalist. News organizations are sometimes blacklisted after breaking an embargo.

History

Ivy Lee, known as the father of modern public relations, was the first to make a press release. The press release was made in October of 1906. The first press release, covered by Lee, was of a railroad accident involving the Pennsylvania Railroad. The accident caused the death of fifty people in Atlantic City, New

Jersey (known as the Atlantic City train wreck) . Lee documented the accident and gave out reports to fellow reporters. The biggest turning point was the honesty that Lee wrote regarding the accident and how truthful it was. Lee's words were so impactful and precise that the New York Times distributed his exact statement and observations. Lee was, and still is, one of the biggest influences and front runners in public relations and press releases. On the account of Lee, press releases have evolved into a necessity for key details among companies to disclose to the public. Since then, press releases have been used to inform other journalists, PR's, and other media relation people of important events, statistics, and announcements.

Other sources

- Electronic press kit (EPK)
- List of press release agencies
- Mat release
- News conference
- Spokesman
- Press service
- Submission software

Special education in the United Kingdom

Special educational needs (**SEN**), also known as **special educational needs and disabilities** (**SEND**) in the United Kingdom refers to the education of children with disabilities.

Definition

The definition of SEN is set out in the Education Act 1996 and was amended in the Special Educational Needs and Disability Bill of 2001. Currently, a child or young person is considered to have SEN if they have a disability or learning difficulty that means they need special educational provision. Special educational provision means that the child needs support that would not generally be provided to a child of the same age in a mainstream school.

Some examples of SEN include:

- A condition which affects behaviour or social skills, such as ADHD or autism
- A condition that affects the ability to read and write, such as dyslexia or another specific learning difficulty
- A condition which affects the ability to learn, such as a learning disability
- A physical impairment, including a visual impairment, hearing impairment, a chronic health condition or poor mobility.

Support available

There are numerous types of support available depending on the child or young person's disability. Some support offered includes:

- Following a different learning programme from the rest of the class
- Extra help from a teaching assistant or the class teacher
- Extra supervision in the classroom or at break time
- Working in a smaller group
- Support to communicate with other pupils
- Help with personal care (such as eating or using the toilet)
- Encouragement to complete tasks the pupil struggles with

Public examinations

Some support available for children with SEN include:

- Extra time to complete the examination
- Rest breaks
- Alternative formats for exam papers
- Use of a reader
- Use of a scribe
- Use of a live speaker for exams that include audio recordings
- Use of a prompter
- Use of a communication professional (a person who can translate questions into British Sign Language or International Sign Language)
- Use of a practical assistant
- Use of a word processor
- Completing the examination in a separate room or venue from other candidates at the school
- Exemption from certain parts of the qualification.

SEN legal regulations

- "SENCO" redirects here. For the geographical place, see Senco.

The SEN systems vary in each nation of the United Kingdom.

England

The current regulations for SEN are set out in the Children and Families Act 2014. Different levels of support are given to children depending on how much support is required. Most children with SEN are given school-level support, known as SEN support. An Education, Health and Care Plan (EHCP) is given to children and young people who are considered to have complex needs. They can be used for children and young people aged 2–25. Children and young people with an EHCP are entitled to a personal budget. Every school must have a Special Educational Needs Co-Ordinator (SENCO), who is responsible for overseeing the support of pupils with SEN. Children with SEN in the UK can attend mainstream or special schools, but legally, local authorities are obliged to educate children in mainstream schools where possible. If a family feels that their child is not receiving sufficient support, they may take their local authority to the Special Educational Needs and Disability Tribunal to appeal any decisions the local authority has made on a child's support.

Local offer

A **Local Offer** (or LO) is a statement detailing the pattern of support which a local authority expects to be available for children and young people with special educational needs (SEN) and/or disabilities within their area. It must include information about education, health and care provision. It should also tell families about training, employment and independent living options available for young people with special educational needs and/or disabilities. In accordance with the SEND Code of Practice, every local authority must publish a Local Offer. The Local Offer or LO should

- provide clear, comprehensive, accessible and up-to-date information about the available provision and how to access it,
- make provision more responsive to local needs and aspirations by directly involving disabled children and those with SEN and their parents, and disabled young people and those with SEN, and service providers in its development and review.

Scotland

In Scotland, the term **additional support needs**is used instead of SEN. As well as children with disabilities, this also encompasses children who may need support for reasons other than disability, such as children who are being bullied or who are in foster care. The Education (Additional Support for Learning) (Scotland) Act 2004 redefined the law relating to the provision of special education to children with additional needs by establishing a framework for the policies of inclusion and generally practicing the "presumption of mainstreaming" in education. Children with complex needs who require support from external organisationsare given a co-ordinated support plan. Families who are not satisfied with the support given are entitled to take the education authority Additional Support Needs for Scotland Tribunal.

Northern Ireland

Regulations for SEN in Northern Ireland are currently governed by the Special Educational Needs and Disability Act (Northern Ireland) 2016. In Northern Ireland, there are five stages of SEN support. Stages 1 to 3 are known as school-based stages. Stage 1 is when concerns are first raised about a child having SEN, and support is given within the classroom, such as differentiated work or different teaching strategies. If the child's difficulties improve at this stage, the child is no longer classed as having SEN. However, if they do not improve, the child will be moved to stage 2. At stage 2, advice from the child's GP or the school doctor is sought and an education plan is drawn up by the SENCO, which describes the difficulties the child has and the support they need. If the child does not make good progress at stage 2, they move on to stage 3.

At stage 3, external specialists, such as educational psychologists are involved in the child's support. If a child does not make progress while on stage 3, they are referred to stage 4. Stage 4 is also known as Statutory Assessment.

Children who have very significant disabilities are referred straight to Statutory Assessment without having to go through the school-based stages. Stage 5 is when a SEN statement is issued. The SEN statement sets out the child's difficulties and the support they require, as well as which school the child should attend (this can be a mainstream or special school).

History

Local authorities became responsible for the education of Deaf children and blind children in 1893. The education of children with disabilities became mandatory in the Education Act 1918. The prevailing attitude at the time was that disabled children should be sent to residential schools rather than attending mainstream schools. The 1944 Education Act created provision for children with disabilities to receive "special educational treatment" in special schools. Children were required to have a medical assessment to be eligible for this. Some children were classified as uneducable, and were not required to attend school. The 1970 Education (Handicapped Children) Act removed uneducable category, which allowed all disabled children to receive an education. SEN statements were introduced in 1978 and parents of children with disabilities were given the right to appeal decisions made by local authorities about decisions on their child's education The 1981 Education Act stated that children should be taught in mainstream schools whenever possible. The 1993 Education Act set out guidelines for identifying pupils with SEN and assessing their needs. The 2001 Special Educational Needs and Disability Act outlawed discrimination against disabled pupils in schools, colleges and other education settings. It also introduced the Special Educational Needs and Disability Tribunal.

Prior to the Children and Families Act 2014, there were three levels of support in England and Wales:

- **school action**- for pupils with relatively low-level needs who can be supported with additional support provided within school, such as the use of specialist teaching materials in lessons.

- **school action plus**- for pupils who need additional support from an external support service. For example, a speech and language therapist or an educational psychologist.
- **SEN statement**-for pupils with more complex needs.

In the English law case of *Skipper v Calderdale Metropolitan Borough School* (2006) EWCA Civ 238, the Court of Appeal allowed the appellant could claim against her former school for failing to diagnose and treat her Dyslexia.

Criticisms

Underfunding

Funding provision for pupils with Special Education Needs and Disabilities, (SEND) is inadequate and as of 2018 £536 million more was needed from the central government. Many parents of SEND children are complaining that their children are not getting the education they need and some have taken legal action to try and force councils to provide for their SEND children. Councils are unable to carry out their statutory duties towards SEND children due to lack of funding from the central government. Educators also complain that they cannot educate SEND pupils as effectively as they would like due to lack of funding. Councils complain they are overstretched due to rising demand and insufficient funding. Antoinette Bramble of the Local Government Association said, “We face a looming crisis in meeting the unprecedented rise in demand for support from children with special educational needs and disabilities. Parents rightly expect and aspire to see that their child has the best possible education and receives the best possible support. Councils have pulled out all the stops to try and do this but are reaching the point where the money is simply not there to keep up with demand." General education spending is also severely stretched making it hard for councils or schools to fund SEND provision out of the general education budget.

In 2019 the Education Select Committee of the House of Commons published a report stating reforms introduced in 2014 had been badly implemented damaging many SEND pupils. Children had to do without support they needed, which affected their mental health as well as their education, children suffered anxiety, depression and self har, children as young as nine had attempted suicide. Children's families had to try and cope with a bureaucracy. The report also criticised a funding shortfall and called for greater accountability in the system. More rigorous inspection systems were called for together with clear consequences following failure. Parents and schools should be able to appeal directly to the DfE if Local Authorities did not meet their legal obligations. School inspections should focus more on SEND, social care ombudsmen and Local Authorities should have greater powers. Robert HalfonMP said, "The DfE cannot continue with a piecemeal and reactive approach to supporting children with Send. Rather than making do with sticking plasters, what is needed is a transformation, a more strategic oversight and fundamental change to ensure a generation of children is no longer let down." Kevin Courtney of the National Education Union said, "Schools and local authorities want to provide the best possible support for SEND pupils, but the tools needed are generally no longer available due to cuts to local services." The Local Government Association stated, "Councils support the reforms set out in the Children and Families Act in 2014, but we were clear at the time that the cost of implementing them had been underestimated by the government."

In the UK local authorities have been cutting special needs provision for children due to austerity. Some children fail to attend school due to severe anxiety, ADHD, autism and similar problems. These children are not getting the special provision they need, they are not getting a diagnosis. Instead the children are treated as truants and their parents are taken to court. A group of parents are mounting a legal challenge to this.

Exclusions and off-rolling

Children with SEN are much more likely to be formally excluded from school or off-rolled. Off-rolling is where a pupil is removed from a school's register, often shortly before GCSEs are due to be taken, which can cause the child's education to be discontinued. There have been claims that children with SEN who are unlikely to achieve the national target of five GCSEs at grades 4 to 9 are being excluded or off-rolled to raise a school's position in league tables. Anne Longfield, the children's commissioner, said "I have become more and more convinced that some schools are seeking to improve their overall exam results by removing vulnerable children from the school roll...sadly this can include children with Send, who have no option but to go into inappropriate alternative provision or home education."

Over-identification

There have been claims that affluent families will push for their child to be identified as having SEN so that the child can access additional support when the child may not genuinely have any disability. The number of children identified as having SEN has increased. Figures published in 2009 showed that 17.8% of pupils in English schools have SEN an increase from 14.9% in 2005, leading to claims that schools are labelling too many children as having SEN. Lorraine Petersen, the former chief executive of the National Association of Special Educational Needs, has said “they [parents] feel a label will give the child and perhaps the family additional support that they may not get without it; access to benefits, for instance, or support with exams or a place in a specialist setting.” In other cases, schools have been accused of identifying non-disabled children as having SEN to hide poor teaching standards.

Under-identification

Conversely, some people argue that there is a problem with children with disabilities not being identified as needing additional support. This is said to be especially difficult for low-income families, who may not be able to afford private diagnostic assessments for conditions such as dyslexia. Bernadette John, the SEN director of *The Good Schools Guide*, says: "There’s a good reason why middle-class parents are better able to get a special needs diagnosis for their child: cash. There is a dire shortage of educational psychologists in local authorities, and children can expect a wait of at least a year to see one for a diagnosis."

Witness statement

A **witness statement** is a signed document recording the evidence of a witness. A definition used in England and Wales is "a written statement signed by a person which contains the evidence which that person would be allowed to give orally".

The United States Federal Rules of Criminal Procedure defines a witness statement as: "(1) a written statement that the witness makes and signs, or otherwise adopts or approves; (2) a substantially verbatim, contemporaneously recorded recital of the witness's oral statement that is contained in any recording or any transcription of a recording; or (3) the witness's statement to a grand jury, however taken or recorded, or a transcription of such a statement."

Financial statement

Financial statements (or **financial reports**) are formal records of the financial activities and position of a business, person, or other entity.

Relevant financial information is presented in a structured manner and in a form which is easy to understand. They typically include four basic financial statements accompanied by a management discussion and analysis:

- A balance sheet or **statement of financial position**, reports on a company's assets, liabilities, and owners equity at a given point in time.

- An income statement—or **profit and loss report** (**P&L report**), or **statement of comprehensive income**, or **statement of revenue & expense**—reports on a company's income, expenses, and profits over a stated period. A profit and loss statement provides information on the operation of the enterprise. These include sales and the various expenses incurred during the stated period.
- A statement of changes in equity or **statement of equity**, or **statement of retained earnings**, reports on the changes in equity of the company over a stated period.
- A cash flow statement reports on a company's cash flow activities, particularly its operating, investing and financing activities over a stated period.
- A comprehensive income statement involves those other comprehensive income items which are not included while determining net income.

(Notably, a balance sheet represents a *single point in time*, where the income statement, the statement of changes in equity, and the cash flow statement each represent activities over a stated *period*.)

For large corporations, these statements may be complex and may include an extensive set of **footnotes to the financial statements** and **management discussion and analysis**. The notes typically describe each item on the balance sheet, income statement and cash flow statement in further detail. Notes to financial statements are considered an integral part of the financial statements.

Purpose for financial statements

"The objective of financial statements is to provide information about the financial position, performance and changes in financial position of an enterprise that is useful to a wide range of users in making economic decisions." Financial statements should be understandable, relevant, reliable and comparable. Reported assets, liabilities, equity, income and expenses are directly related to an organization's financial position.

Financial statements are intended to be understandable by readers who have "a reasonable knowledge of business and economic activities and accounting and who are willing to study the information diligently." Financial statements may be used by users for different purposes:

- Owners and managers require financial statements to make important business decisions that affect its continued operations. Financial analysisis then performed on these statements to provide management with a more detailed understanding of the figures. These statements are also used as part of management's annual report to the stockholders.
- Employees also need these reports in making collective bargaining agreements (CBA) with the management, in the case of labor unions or for individuals in discussing their compensation, promotion and rankings.
- Prospective investors make use of financial statements to assess the viability of investing in a business. Financial analyses are often used by

investors and are prepared by professionals (financial analysts), thus providing them with the basis for making investment decisions.
- Financial institutions (banks and other lending companies) use them to decide whether to grant a company with fresh working capital or extend debt securities (such as a long-term bank loan or debentures) to finance expansion and other significant expenditures.

Consolidated

Consolidated financial statements are defined as "Financial statements of a group in which the assets, liabilities, equity, income, expenses and cash flows of the parent (company) and its subsidiaries are presented as those of a single economic entity", according to International Accounting Standard 27 "Consolidated and separate financial statements", and International Financial Reporting Standard 10 "Consolidated financial statements".

Government

The rules for the recording, measurement and presentation of government financial statements may be different from those required for business and even for non-profit organizations. They may use either of two accounting methods: accrual accounting, or cost accounting, or a combination of the two (OCBOA). A complete set of chart of accountsis also used that is substantially different from the chart of a profit-oriented business.

Personal

Personal financial statements may be required from persons applying for a personal loan or financial aid. Typically, a personal financial statement consists of a single form for reporting personally held assets and liabilities (debts), or personal sources of income and expenses, or both. The form to be filled out is determined by the organization supplying the loan or aid.

Audit and legal implications

Although laws differ from country to country, an audit of the financial statements of a public company is usually required for investment, financing, and tax purposes. These are usually performed by independent accountants or auditing firms. Results of the audit are summarized in an audit report that either provide an unqualified opinion on the financial statements or qualifications as to its fairness and accuracy. The audit opinion on the financial statements is usually included in the annual report.

There has been much legal debate over who an auditor is liable to. Since audit reports tend to be addressed to the current shareholders, it is commonly thought that they owe a legal duty of care to them. But this may not be the case as determined by common law precedent. In Canada, auditors are liable only to investors using a prospectus to buy shares in the primary market. In the United Kingdom, they have been held liable to potential investors when the auditor was aware of the potential investor and how they would use the information in the financial statements. Nowadays auditors tend to include in their report liability restricting language, discouraging anyone other than the addressees of their report from relying on it. Liability is an important issue: in the UK, for example, auditors have unlimited liability.

In the United States, especially in the post-Enron era there has been substantial concern about the accuracy of financial statements. Corporate officers—the chief executive officer (CEO) and chief financial officer (CFO)—are personally responsible for fair financial reporting that provides an accurate sense of the organization to those reading the report.

Standards and regulations

Different countries have developed their own accounting principles over time, making international comparisons of companies difficult. To ensure uniformity and comparability between financial statements prepared by different companies, a set of guidelines and rules are used. Commonly referred to as Generally Accepted Accounting Principles (GAAP), these set of guidelines provide the basis in the preparation of financial statements, although many companies voluntarily disclose information beyond the scope of such requirements.

Recently there has been a push towards standardizing accounting rules made by the International Accounting Standards Board ("IASB"). IASB develops International Financial Reporting Standards that have been adopted by Australia, Canada and the European Union (for publicly quoted companies only), are under consideration in South Africa and other countries. The United StatesFinancial Accounting

Standards Board has made a commitment to converge the U.S. GAAP and IFRS over time.

Inclusion in annual reports

To entice new investors, public companies assemble their financial statements on fine paper with pleasing graphics and photos in an annual report to shareholders, attempting to capture the excitement and culture of the organization in a "marketing brochure" of sorts. Usually the company's chief executive will write a letter to shareholders, describing management's performance and the company's financial highlights.

In the United States, prior to the advent of the internet, the annual report was considered the most effective way for corporations to communicate with individual shareholders. Blue chip companies went to great expense to produce and mail out attractive annual reports to every shareholder. The annual report was often prepared in the style of a coffee table book.

Notes

Additional information added to the end of financial statements that help explain specific items in the statements as well as provide a more comprehensive assessment of a company's financial condition are known as notes (or "notes to financial statements").

Notes to financial statements can include information on debt, accounts, contingent liabilities, on going concern criteria, or on contextual information explaining the financial numbers (e.g. to indicate a lawsuit). The notes clarify individual statement line-items. Notes are also used to explain the accounting methods used to prepare the statements and they support valuations for how particular accounts have been computed. As an example: If a company lists a loss on a fixed asset impairment line in their income statement, the notes may state the reason for the impairment by describing how the asset became impaired.

In consolidated financial statements, all subsidiariesare listed as well as the amount of ownership (controlling interest) that the parent company has in the subsidiaries.

Any items within the financial statements that are valuated by estimation are part of the notes if a substantial difference exists between the amount of the estimate previously reported and the actual result. Full disclosure of the effects of the differences between the estimate and actual results should be included.

Management discussion and analysis

Management discussion and analysis or MD&A is an integrated part of a company's annual financial statements. The purpose of the MD&A is to provide a narrative explanation, through the eyes of management, of how an entity has performed in the past, its financial condition, and its future prospects. In so doing, the MD&A attempt to provide investors with complete, fair, and balanced information to help them decide whether to invest or continue to invest in an entity.

The section contains a description of the year gone by and some of the key factors that influenced the business of the company in that year, as well as a fair and unbiased overview of the company's past, present, and future.

MD&A typically describes the corporation's liquidity position, capital resources, results of its operations, underlying causes of material changes in financial statement items (such as asset impairment and restructuring charges), events of unusual or infrequent nature (such as mergers and acquisitions or share buybacks), positive and negative trends, effects of inflation, domestic and international market risks, and significant uncertainties.

Move to electronic statements

Financial statements have been created on paper for hundreds of years. The growth of the Web has seen more and more financial statements created in an electronic form which is exchangeable over the Web. Common forms of electronic financial statements are PDF and HTML. These types of electronic financial statements have their drawbacks in that it still takes a human to read the information in order to reuse the information contained in a financial statement.

More recently a market driven global standard, XBRL (Extensible Business Reporting Language), which can be used for creating financial statements in a structured and computer readable format, has become more popular as a format for creating financial statements. Many regulators around the world such as the U.S. Securities and Exchange Commission have mandated XBRL for the submission of financial information.

The UN/CEFACT created, with respect to Generally Accepted Accounting Principles, (GAAP), internal or external financial reportingXML messages to be used between enterprises and their partners, such as private interested parties (e.g. bank) and public collecting bodies (e.g. taxation authorities). Many regulators use such messages to collect financial and economic information.

Political statement

The term **political statement**is used to refer to any act or non-verbal form of communication that is intended to influence a decision to be made for or by a political party.

A political statement can vary from a mass demonstration to the wearing of a badge with a political slogan. It was a term popularised in the 1960s but still has some currency.

The term has also been used to describe negotiated statements such as the Seville Statement on Violence or the Waldorf Statement, or extempore utterances with political implications.

Chapter 5

BASIC and Assembly Language

BASIC

BASIC (**Beginners' All-purpose Symbolic Instruction Code**) is a family of general-purpose, high-level programming languages whose design philosophy emphasizes ease of use. The original version was designed by John G. Kemeny and Thomas E. Kurtz and released at Dartmouth College in 1964. They wanted to enable students in fields other than science and mathematics to use computers. At the time, nearly all use of computers required writing custom software, which was something only scientists and mathematicians tended to learn.

In addition to the language itself, Kemeny and Kurtz developed the Dartmouth Time Sharing System (DTSS), which allowed multiple users to edit and run BASIC programs at the same time. This general model became very popular on minicomputer systems like the PDP-11 and Data General Nova in the late 1960s and early 1970s. Hewlett-Packard produced an entire computer line for this method of operation, introducing the HP2000 series in the late 1960s and continuing sales into the 1980s. Many early video games trace their history to one of these versions of BASIC.

The emergence of early microcomputers in the mid-1970s led to the development of a number of BASIC dialects, including Microsoft BASIC in 1975. Due to the tiny main memory available on these machines, often 4 KB, a variety of Tiny

BASIC dialects was also created. BASIC was available for almost any system of the era, and naturally became the *de facto* programming language for the home computer systems that emerged in the late 1970s. These machines almost always had a BASIC interpreter installed by default, often in the machine's firmware or sometimes on a ROM cartridge.

BASIC fell from use in the early 1990s, as newer machines with far greater capabilities came to market and other programming languages (such as Pascal and C) became tenable. In 1991, Microsoft released Visual Basic, combining a greatly updated version of BASIC with a visual forms builder. This reignited use of the language and "VB" remains a major programming language in the form of VB.NET.

Origin

John G. Kemeny was the math department chairman at Dartmouth College. Based largely on his reputation as an innovator in math teaching, in 1959 the school won an Alfred P. Sloan Foundation award for $500,000 to build a new department building. Thomas E. Kurtz had joined the department in 1956, and from the 1960s Kemeny and Kurtz agreed on the need for programming literacy among students outside the traditional STEM fields. Kemeny later noted that "Our vision was that every student on campus should have access to a computer, and any faculty member should be able to use a computer in the classroom whenever appropriate. It was as simple as that."

Kemeny and Kurtz had made two previous experiments with simplified languages, DARSIMCO (Dartmouth Simplified Code) and DOPE (Dartmouth Oversimplified Programming Experiment). These did not progress past a single freshman class. New experiments using Fortran and ALGOL followed, but Kurtz concluded these languages were too tricky for what they desired. As Kurtz noted, Fortran had numerous oddly-formed commands, notably an "almost impossible-to-memorize convention for specifying a loop: 'DO 100, I = 1, 10, 2'. Is it '1, 10, 2' or '1, 2, 10', and is the comma after the line number required or not?"

Moreover, the lack of any sort of immediate feedback was a key problem; the machines of the era used batch processing and took a long time to complete a run of a program. While Kurtz was visiting MIT, John McCarthy suggested that time-sharing offered a solution; a single machine could divide up its processing time among many users, giving them the illusion of having a (slow) computer to themselves. Small programs would return results in a few seconds. This led to increasing interest in a system using time-sharing and a new language specifically for use by non-STEM students.

Kemeny wrote the first version of BASIC. The acronym*BASIC* comes from the name of an unpublished paper by Thomas Kurtz. The new language was heavily patterned on FORTRAN II; statements were one-to-a-line, numbers were used to indicate the target of loops and branches, and many of the commands were similar or identical to Fortran. However, the syntaxwas changed wherever it could be improved. For instance, the difficult to remember `DO` loop was replaced by the much easier to remember `FOR I = 1 TO 10 STEP 2`, and the line number used in the DO was instead indicated by the `NEXT I`. Likewise, the cryptic `IF` statement of Fortran, whose syntax matched a particular instruction of the machine on which it was originally written, became the simpler `IF I=5 THEN GOTO 100`. These changes made the language much less idiosyncratic while still having an overall structure and feel similar to the original FORTRAN.

The project received a $300,000 grant from the National Science Foundation, which was used to purchase a GE-225 computer for processing, and a Datanet-30 realtime processor to handle the Teletype Model 33teleprinters used for input and output. A team of a dozen undergraduates worked on the project for about a year, writing both the DTSS system and the BASIC compiler. The first version BASIC language was released on 1 May 1964. One of the graduate students on the implementation team was Mary Kenneth Keller, one of the first people in the United States to earn a Ph.D. in computer science and the first woman to do so.

Initially, BASIC concentrated on supporting straightforward mathematical work, with matrix arithmetic support from its initial implementation as a batch language, and character string functionality being added by 1965. Usage in the university rapidly expanded, requiring the main CPU to be replaced by a GE-235, and still later by a GE-635. By the early 1970s there were hundreds of terminals connected to the machines at Dartmouth, some of them remotely.

Wanting use of the language to become widespread, its designers made the compiler available free of charge. In the 1960s, software became a chargeable commodity; until then, it was provided without charge as a service with the very expensive computers, usually available only to lease. They also made it available to high schools in the Hanover, New Hampshire area and regionally throughout New England on Teletype Model 33 and Model 35 teleprinter terminals connected to Dartmouth via dial-up phone lines, and they put considerable effort into promoting the language. In the following years, as other dialects of BASIC appeared, Kemeny and Kurtz's original BASIC dialect became known as *Dartmouth BASIC*.

New Hampshire recognized the accomplishment in 2019 when it erected a highway historical marker in Hanover describing creation of "the first user-friendly programming language".

Spread on time-sharing services

The emergence of BASIC took place as part of a wider movement towards time-sharing systems. First conceptualized during the late 1950s, the idea became so dominant in the computer industry by the early 1960s that its proponents were speaking of a future in which users would "buy time on the computer much the same way that the average household buys power and water from utility companies".

General Electric, having worked on the Dartmouth project, wrote their own underlying operating system and launched an online time-sharing system known as Mark I. It featured BASIC as one of its primary selling points. Other companies in the emerging field quickly followed suit; Tymshare introduced SUPER BASIC in 1968, CompuServe had a version on the DEC-10 at their launch in 1969, and by the early 1970s BASIC was largely universal on general-purpose mainframe computers.

Even IBM eventually joined the club with the introduction of VS-BASIC in 1973.

Although time-sharing services with BASIC were successful for a time, the widespread success predicted earlier was not to be. The emergence of minicomputers during the same period, and especially low-cost microcomputers in the mid-1970s, allowed anyone to purchase and run their own systems rather than buy online time which was typically billed at dollars per minute.

Spread on minicomputers

BASIC, by its very nature of being small, was naturally suited to porting to the minicomputer market, which was emerging at the same time as the time-sharing services. These machines had very small main memory, perhaps as little as 4 KB in modern terminology, and lacked high-performance storage like hard drives that make compilers practical. On these systems, BASIC was normally implemented as an interpreter rather than a compiler due to the reduced need for working memory.

A particularly important example was HP Time-Shared BASIC, which, like the original Dartmouth system, used two computers working together to implement a time-sharing system. The first, a low-end machine in the HP 2100 series, was used to control user input and save and load their programs to tape or disk. The other, a high-end version of the same underlying machine, ran the programs and generated output. For a cost of about $100,000, one could own a machine capable of running between 16 and 32 users at the same time. The system, bundled as the HP 2000, was the first mini platform to offer time-sharing and was an immediate runaway success, catapulting HP to become the third-largest vendor in the minicomputer space, behind DEC and Data General (DG).

DEC, the leader in the minicomputer space since the mid-1960s, had initially ignored BASIC. This was due to their work with RAND Corporation, who had purchased a PDP-6 to run their JOSS language, which was conceptually very similar to BASIC. This led DEC to introduce a smaller, cleaned up version of JOSS known as FOCAL, which they heavily promoted in the late 1960s. However, with timesharing systems widely offering BASIC, and all of their competition in the minicomputer space doing the same, DEC's customers were clamoring for BASIC. After management repeatedly ignored their pleas, David H. Ahl took it upon himself to buy a BASIC for the PDP-8, which was a major success in the education market. By the early 1970s, FOCAL and JOSS had been forgotten and BASIC had become almost universal in the minicomputer market. DEC would go on to introduce their updated version, BASIC-PLUS, for use on the RSTS/E time-sharing operating system.

During this period a number of simple text-based games were written in BASIC, most notably Mike Mayfield's *Star Trek*. David Ahl collected these, some ported from FOCAL, and published them in an educational newsletter he compiled. He later collected a number of these into book form, *101 BASIC Computer Games*, published in 1973. During the same period, Ahl was involved in the creation of a small computer for education use, an early personal computer. When management refused to support the concept, Ahl left DEC in 1974 to found the seminal computer magazine, *Creative Computing*. The book remained popular, and was re-published on several occasions.

Explosive growth: the home computer era

The introduction of the first microcomputers in the mid-1970s was the start of explosive growth for BASIC. It had the advantage that it was fairly well known to the young designers and computer hobbyists who took an interest in microcomputers, many of whom had seen BASIC on minis or mainframes. Despite Dijkstra's famous judgement in 1975, "It is practically impossible to teach good programming to students that have had a prior exposure to BASIC: as potential programmers they are mentally mutilated beyond hope of regeneration", BASIC was one of the few languages that was both high-level enough to be usable by those without training and small enough to fit into the microcomputers of the day, making it the *de facto* standard programming language on early microcomputers.

The first microcomputer version of BASIC was co-written by Bill Gates, Paul Allen and Monte Davidoff for their newly formed company, Micro-Soft. This was released by MITS in punch tape format for the Altair 8800 shortly after the machine itself, immediately cementing BASIC as the primary language of early microcomputers. Members of the Homebrew Computer Club began circulating copies of the program, causing Gates to write his Open Letter to Hobbyists, complaining about this early example of software piracy.

Partially in response to Gates's letter, and partially to make an even smaller BASIC that would run usefully on 4 KB machines, Bob Albrecht urged Dennis Allison to write their own variation of the language. How to design and implement a stripped-down version of an interpreter for the BASIC language was covered in articles by Allison in the first three quarterly issues of the *People's Computer Company* newsletter published in 1975 and implementations with source code published in *Dr. Dobb's Journal of Tiny BASIC Calisthenics & Orthodontia: Running Light Without Overbyte*. This led to a wide variety of Tiny BASICs with added features or other improvements, with versions from Tom Pittman and Li-Chen Wangbecoming particularly well known.

Micro-Soft, by this time Microsoft, ported their interpreter for the MOS 6502, which quickly become one of the most popular microprocessors of the 8-bit era. When new microcomputers began to appear, notably the "1977 trinity" of the TRS-80, Commodore PET and Apple II, they either included a version of the MS code, or quickly introduced new models with it. By 1978, MS BASIC was a *de facto* standard and practically every home computer of the 1980s included it in ROM. Upon boot, a BASIC interpreter in direct modewas presented.

Commodore Business Machines included Commodore BASIC, based on Microsoft BASIC. The Apple II and TRS-80 each had two versions of BASIC, a smaller introductory version introduced with the initial releases of the machines and an MS-based version introduced as interest in the platforms increased. As new companies entered the field, additional versions were added that subtly changed the BASIC family. The Atari 8-bit family had its own Atari BASIC that was modified in order to fit on an 8 KB ROM cartridge. Sinclair BASICwas introduced in 1980 with the Sinclair ZX80, and was later extended for the Sinclair ZX81 and the Sinclair ZX Spectrum.

The BBC published BBC BASIC, developed by Acorn Computers Ltd, incorporating many extra structured programming keywords and advanced floating-point operation features.

As the popularity of BASIC grew in this period, computer magazines published complete source code in BASIC for video games, utilities, and other programs. Given BASIC's straightforward nature, it was a simple matter to type in the code from the magazine and execute the program. Different magazines were published featuring programs for specific computers, though some BASIC programs were considered universal and could be used in machines running any variant of BASIC (sometimes with minor adaptations). Many books of type-in programs were also available, and in particular, Ahl published versions of the original 101 BASIC games converted into the Microsoft dialect and published it from *Creative Computing* as *BASIC Computer Games*. This book, and its sequels, provided hundreds of ready-to-go programs that could be easily converted to practically any BASIC-running platform. The book reached the stores in 1978, just as the home computer market was starting off, and it became the first million-selling computer book. Later packages, such as Learn to Program BASIC would also have gaming as an introductory focus. On the business-focused CP/Mcomputers which soon became widespread in small business environments, Microsoft BASIC (MBASIC) was one of the leading applications.

In 1978, David Lien published the first edition of *The BASIC Handbook: An Encyclopedia of the BASIC Computer Language*, documenting keywords across over 78 different computers. By 1981, the second edition documented keywords from over 250 different computers, showcasing the explosive growth of the microcomputer era.

IBM PC and compatibles

When IBM was designing the IBM PC they followed the paradigm of existing home computers in wanting to have a built-in BASIC. They sourced this from Microsoft – IBM Cassette BASIC – but Microsoft also produced several other versions of BASIC for MS-DOS/PC DOS including IBM Disk BASIC (BASIC D), IBM BASICA (BASIC A), GW-BASIC (a BASICA-compatible version that did not need IBM's ROM) and QBasic, all typically bundled with the machine. In addition they produced the Microsoft BASIC Compiler aimed at professional programmers. Turbo Pascal-publisher Borland published Turbo Basic 1.0 in 1985 (successor versions are still being marketed by the original author under the name PowerBASIC). Microsoft wrote the windowed AmigaBASIC that was supplied with version 1.1 of the pre-emptive multitasking GUI Amiga computers (late 1985 / early 1986), although the product unusually did not bear any Microsoft marks.

These later variations introduced many extensions, such as improved string manipulation and graphics support, access to the file system and additional data types. More important were the facilities for structured programming, including additional control structures and proper subroutines supporting local variables. However, by the latter half of the 1980s, users were increasingly using pre-made applications written by others rather than learning programming themselves; while professional programmers now had a wide range of more advanced languages available on small computers. C and later

C++ became the languages of choice for professional "shrink wrap" application development.

Visual Basic

In 1991, Microsoft introduced Visual Basic, an evolutionary development of QuickBASIC. It included constructs from that language such as block-structured control statements, parameterized subroutines and optional static typing as well as object-oriented constructs from other languages such as "With" and "For Each". The language retained some compatibility with its predecessors, such as the Dim keyword for declarations, "Gosub"/Return statements and optional line numbers which could be used to locate errors. An important driver for the development of Visual Basic was as the new macro language for Microsoft Excel, a spreadsheet program. To the surprise of many at Microsoft who still initially marketed it as a language for hobbyists, the language came into widespread use for small custom business applications shortly after the release of VB version 3.0, which is widely considered the first relatively stable version.

While many advanced programmers still scoffed at its use, VB met the needs of small businesses efficiently as by that time, computers running Windows 3.1 had become fast enough that many business-related processes could be completed "in the blink of an eye" even using a "slow" language, as long as large amounts of data were not involved. Many small business owners found they could create their own small, yet useful applications in a few evenings to meet their own specialized needs. Eventually, during the lengthy lifetime of VB3, knowledge of Visual Basic had become a marketable job skill.

Microsoft also produced VBScript in 1996 and Visual Basic .NET in 2001. The latter has essentially the same power as C# and Java but with syntax that reflects the original Basic language. The IDE, with its event-drivenGUI builder, was also influential on other tools, most notably Borland Software's Delphi for Object Pascal and its own descendants such as Lazarus.

Mainstream support for the final version 6.0 of the original Visual Basic ended on March 31, 2005, followed by extended support in March 2008. On March 11, 2020, Microsoft announced that evolution of the VB.NET language had also concluded, although it was still supported. Meanwhile, competitors exist such as Xojo and Gambas.

Post-1990 versions and dialects

Many other BASIC dialects have also sprung up since 1990, including the open sourceQB64 and FreeBASIC, inspired by QBasic, and the Visual Basic-styled RapidQ, Basic ForQt and Gambas. Modern commercial incarnations include PureBasic, PowerBASIC, Xojo, Monkey X and True BASIC (the direct successor to Dartmouth BASIC from a company controlled by Kurtz).

Several web-based simple BASIC interpreters also now exist, including Microsoft's Small Basic. Many versions of BASIC are also now available for smartphones and tablets via the Apple App Store, or Google Play store for Android. On game consoles, an application for the Nintendo 3DS and Nintendo DSi called *Petit Computer* allows for programming in a slightly modified version of BASIC with DS button support. A version has also been released for Nintendo Switch.

Calculators

Variants of BASIC are available on graphing and otherwise programmable calculators made by Texas Instruments, HP, Casio, and others.

Windows command-line

QBasic, a version of Microsoft QuickBASIC without the linker to make EXE files, is present in the Windows NT and DOS-Windows 95 streams of operating systems and can be obtained for more recent releases like Windows 7 which do not have them. Prior to DOS 5, the Basic interpreter was GW-Basic. QuickBasic is part of a series of three languages issued by Microsoft for the home and office power user and small-scale professional development; QuickC and QuickPascal are the other two. For Windows 95 and 98, which do not have QBasic installed by default, they can be copied from the installation disc, which will have a set of directories for old and optional software; other missing commands like Exe2Bin and others are in these same directories.

Other

The various Microsoft, Lotus, and Corel office suites and related products are programmable with Visual Basic in one form or another, including LotusScript, which is very similar to

VBA 6. The Host Explorer terminal emulator uses WWB as a macro language; or more recently the programme and the suite in which it is contained is programmable in an in-house Basic variant known as Hummingbird Basic. The VBScript variant is used for programming web content, Outlook 97, Internet Explorer, and the Windows Script Host. WSH also has a Visual Basic for Applications (VBA) engine installed as the third of the default engines along with VBScript, JScript, and the numerous proprietary or open source engines which can be installed like PerlScript, a couple of Rexx-based engines, Python, Ruby, Tcl, Delphi, XLNT, PHP, and others; meaning that the two versions of Basic can be used along with the other mentioned languages, as well as LotusScript, in a WSF file, through the component object model, and other WSH and VBA constructions. VBScript is one of the languages that can be accessed by the 4Dos, 4NT, and Take Command enhanced shells. SaxBasic and WWB are also very similar to the Visual Basic line of Basic implementations. The pre-Office 97 macro language for Microsoft Word is known as WordBASIC. Excel 4 and 5 use Visual Basic itself as a macro language. Chipmunk Basic, an old-school interpreter similar to BASICs of the 1970s, is available for Linux, Microsoft Windows and macOS.

Legacy

The ubiquity of BASIC interpreters on personal computers was such that textbooks once included simple "Try It In BASIC" exercises that encouraged students to experiment with mathematical and computational concepts on classroom or home computers. Popular computer magazines of the day typically included type-in programs.

Futurist and sci-fi writer David Brin mourned the loss of ubiquitous BASIC in a 2006 *Salon*article as have others who first used computers during this era. In turn, the article prompted Microsoft to develop and release Small Basic; it also inspired similar projects like Basic-256. Dartmouth held a 50th anniversary celebration for BASIC on 1 May 2014, as did other organisations; at least one organisation of VBA programmers organised a 35th anniversary observance in 1999.

Dartmouth College celebrated the 50th anniversary of the BASIC language with a day of events on April 30, 2014. A short documentary film was produced for the event.

Syntax

Typical BASIC keywords

Data manipulation

- `LET`
- assigns a value (which may be the result of an expression) to a variable. In most dialects of BASIC, `LET` is optional, and a line with no other identifiable keyword will assume the keyword to be `LET`.
- `DATA`
- holds a list of values which are assigned sequentially using the READ command.
- `READ`
- reads a value from a `DATA` statement and assigns it to a variable. An internal pointer keeps track of the last

DATA element that was read and moves it one position forward with each READ.

- RESTORE
- resets the internal pointer to the first DATA statement, allowing the program to begin READing from the first value.

Program flow control

- IF ... THEN ... {ELSE}
- used to perform comparisons or make decisions. ELSE was not widely supported, especially in earlier versions.
- FOR ... TO ... {STEP} ... NEXT
- repeat a section of code a given number of times. A variable that acts as a counter is available within the loop.
- WHILE ... WEND and REPEAT ... UNTIL
- repeat a section of code while the specified condition is true. The condition may be evaluated before each iteration of the loop, or after. Both of these commands are found mostly in later dialects.
- DO ... LOOP {WHILE} or {UNTIL}
- repeat a section of code indefinitely or while/until the specified condition is true. The condition may be evaluated before each iteration of the loop, or after. Similar to WHILE, these keywords are mostly found in later dialects.
- GOTO
- jumps to a numbered or labelled line in the program.
- GOSUB

- jumps to a numbered or labelled line, executes the code it finds there until it reaches a `RETURN` command, on which it jumps back to the statement following the `GOSUB`, either after a colon, or on the next line. This is used to implement subroutines.
- `ON ... GOTO/GOSUB`
- chooses where to jump based on the specified conditions. See Switch statement for other forms.
- `DEF FN`
- a pair of keywords introduced in the early 1960s to define functions. The original BASIC functions were modelled on FORTRAN single-line functions. BASIC functions were one expression with variable arguments, rather than subroutines, with a syntax on the model of `DEF FND(x) = x*x` at the beginning of a program. Function names were originally restricted to FN, plus one letter, *i.e.*, FNA, FNB ...

Input and output

- `LIST`
- displays the full source code of the current program.
- `PRINT`
- displays a message on the screen or other output device.
- `INPUT`
- asks the user to enter the value of a variable. The statement may include a prompt message.
- `TAB`
- used with `PRINT` to set the position where the next character will be shown on the screen or printed on paper. `AT` is an alternative form.

- `SPC`
- prints out a number of space characters. Similar in concept to `TAB` but moves by a number of additional spaces from the current column rather that moving to a specified column.

Mathematical functions

- `ABS`
- Absolute value
- `ATN`
- Arctangent (result in radians)
- `COS`
- Cosine (argument in radians)
- `EXP`
- Exponential function
- `INT`
- Integer part (typically floor function)
- `LOG`
- Natural logarithm
- `RND`
- Random number generation
- `SIN`
- Sine (argument in radians)
- `SQR`
- Square root
- `TAN`
- Tangent (argument in radians)

Miscellaneous

- `REM`

- holds a programmer's comment or REMark; often used to give a title to the program and to help identify the purpose of a given section of code.
- `USR`
- transfers program control to a machine language subroutine, usually entered as an alphanumeric string or in a list of DATA statements.
- `CALL`
- alternative form of `USR` found in some dialects. Does not require an artificial parameter to complete the function-like syntax of `USR`, and has a clearly defined method of calling different routines in memory.
- `TRON`
- turns on display of each line number as it is run ("TRace ON"). This was useful for debugging or correcting of problems in a program.
- `TROFF`
- turns off the display of line numbers.
- `ASM`
- some compilers such as Freebasic, Purebasic, and Powerbasic also support inline assembly language, allowing the programmer to intermix high-level and low-level code, typically prefixed with "ASM" or "!" statements.

Data types and variables

Minimal versions of BASIC had only integer variables and one- or two-letter variable names, which minimized requirements of limited and expensive memory (RAM). More powerful versions had floating-point arithmetic, and variables could be labelled with names six or more characters long. There were some problems and restrictions in early implementations; for example, Applesoft BASIC allowed variable names to be several characters long, but only the first two were significant, thus it was possible to inadvertently write a program with variables "LOSS" and "LOAN", which would be treated as being the same; assigning a value to "LOAN" would silently overwrite the value intended as "LOSS". Keywords could not be used in variables in many early BASICs; "SCORE" would be interpreted as "SC" OR "E", where OR was a keyword. String variables are usually distinguished in many microcomputer dialects by having $ suffixed to their name as a sigil, and values are often identified as strings by being delimited by "double quotation marks". Arrays in BASIC could contain integers, floating point or string variables.

Some dialects of BASIC supported matrices and matrix operations, useful for the solution of sets of simultaneous linear algebraic equations. These dialects would directly support matrix operations such as assignment, addition, multiplication (of compatible matrix types), and evaluation of a determinant. Many microcomputer BASICs did not support this data type; matrix operations were still possible, but had to be programmed explicitly on array elements.

Examples

Unstructured BASIC

New BASIC programmers on a home computer might start with a simple program, perhaps using the language's PRINT statement to display a message on the screen; a well-known and often-replicated example is Kernighan and Ritchie's "Hello, World!" program:

```
10PRINT"Hello, World!"
20END
```

An infinite loopcould be used to fill the display with the message:

```
10PRINT"Hello, World!"
20GOTO10
```

Note that the `END` statement is optional and has no action in most dialects of BASIC. It was not always included, as is the case in this example. This same program can be modified to print a fixed number of messages using the common `FOR...NEXT` statement:

```
10LETN=10
20FORI=1TON
30PRINT"Hello, World!"
40NEXTI
```

Most first-generation BASIC versions, such as MSX BASIC and GW-BASIC, supported simple data types, loop cycles, and arrays. The following example is written for GW-BASIC, but will work in most versions of BASIC with minimal changes:

```
10INPUT"What is your name: ";U$
20PRINT"Hello ";U$
30INPUT"How many stars do you want: ";N
40S$=""
50FORI=1TON
60S$=S$+"*"
70NEXTI
80PRINTS$
90INPUT"Do you want more stars? ";A$
100IFLEN(A$)=0THENGOTO90
110A$=LEFT$(A$,1)
120IFA$="Y"ORA$="y"THENGOTO30
130PRINT"Goodbye ";U$
140END
```

The resulting dialog might resemble:

```
What is your name: Mike
Hello Mike
How many stars do you want: 7
*******
Do you want more stars? yes
How many stars do you want: 3
***
Do you want more stars? no
Goodbye Mike
```

The original Dartmouth Basic was unusual in having a matrix keyword, MAT. Although not implemented by most later microprocessor derivatives, it is used in this example from the 1968 manual which averages the numbers that are input:

```
5LETS=0
10MATINPUTV
20LETN=NUM
30IFN=0THEN99
40FORI=1TON
45LETS=S+V(I)
50NEXTI
60PRINTS/N
70GOTO5
99END
```

Structured BASIC

Second-generation BASICs (for example, VAX Basic, SuperBASIC, True BASIC, QuickBASIC, BBC BASIC, Pick BASIC, PowerBASIC, Liberty BASIC and (arguably) COMAL) introduced a number of features into the language, primarily related to structured and procedure-oriented programming. Usually, line numberingis omitted from the language and replaced with labels (for GOTO) and procedures to encourage easier and more flexible design. In addition keywords and structures to support repetition, selection and procedures with local variables were introduced.

The following example is in Microsoft QuickBASIC:

```
REM QuickBASIC example
REM Forward declaration - allows the main code to call a
REM    subroutine that is defined later in the source code
DECLARESUBPrintSomeStars(StarCount!)

REM Main program follows
INPUT"What is your name: ",UserName$
PRINT"Hello ";UserName$
DO
INPUT"How many stars do you want: ",NumStars
CALLPrintSomeStars(NumStars)
DO
INPUT"Do you want more stars? ",Answer$
LOOPUNTILAnswer$<>""
Answer$=LEFT$(Answer$,1)
LOOPWHILEUCASE$(Answer$)="Y"
PRINT"Goodbye ";UserName$
END

REM subroutine definition
SUBPrintSomeStars(StarCount)
REMThisprocedureusesalocalvariablecalledStars$
Stars$=STRING$(StarCount,"*")
PRINTStars$
ENDSUB
```

Object-oriented BASIC

Third-generation BASIC dialects such as Visual Basic, Xojo, Gambas, StarOffice Basic, BlitzMax and PureBasic introduced features to support object-oriented and event-driven programming paradigm. Most built-in procedures and functions are now represented as *methods* of standard objects rather than *operators*. Also, the operating system became increasingly accessible to the BASIC language.

The following example is in Visual Basic .NET:

```
PublicModuleStarsProgram
PrivateFunctionAsk(promptAsString)AsString
Console.Write(prompt)
ReturnConsole.ReadLine()
EndFunction

PublicSubMain()
DimuserName=Ask("What is your name: ")
Console.WriteLine("Hello {0}",userName)

DimanswerAsString
```

```
Do
DimnumStars=CInt(Ask("How many stars do you want: "))
DimstarsAsNewString("*"c,numStars)
Console.WriteLine(stars)

Do
answer=Ask("Do you want more stars? ")
LoopUntilanswer<>""
LoopWhileanswer.StartsWith("Y",StringComparison.OrdinalIgnoreCase)

Console.WriteLine("Goodbye {0}",userName)
EndSub
EndModule
```

Standards

- ANSI/ISO/IEC Standard for Minimal BASIC:
- ANSI X3.60-1978 "For minimal BASIC"
- ISO/IEC 6373:1984 "Data Processing—Programming Languages—Minimal BASIC"
- ECMA-55 Minimal BASIC *(withdrawn, similar to ANSI X3.60-1978)*
- ANSI/ISO/IEC Standard for Full BASIC:
- ANSI X3.113-1987 "Programming Languages Full BASIC"
- INCITS/ISO/IEC 10279-1991 (R2005) "Information Technology – Programming Languages – Full BASIC"
- ANSI/ISO/IEC Addendum Defining Modules:
- ANSI X3.113 Interpretations-1992 "BASIC Technical Information Bulletin # 1 Interpretations of ANSI 03.113-1987"
- ISO/IEC 10279:1991/ Amd 1:1994 "Modules and Single Character Input Enhancement"
- ECMA-116 BASIC *(withdrawn, similar to ANSI X3.113-1987)*

Assembly language

In computer programming, **assembly language** (or **assembler language**), sometimes abbreviated **asm**, is any low-level programming language in which there is a very strong correspondence between the instructions in the language and the architecture'smachine codeinstructions. Because assembly depends on the machine code instructions, every assembly language is designed for exactly one specific computer architecture. Assembly language may also be called*symbolic machine code*.

Assembly code is converted into executable machine code by a utility program referred to as an *assembler*. The conversion process is referred to as *assembly*, as in *assembling* the source code. Assembly language usually has one statement per machine instruction (1:1), but constants, comments, assembler directives, symbolic labels of program and memory locations, and macrosare generally also supported.

The term "assembler" is generally attributed to Wilkes, Wheeler and Gill in their 1951 book *The Preparation of Programs for an Electronic Digital Computer*, who, however, used the term to mean "a program that assembles another program consisting of several sections into a single program".

Each assembly language is specific to a particular computer architecture and sometimes to an operating system. However, some assembly languages do not provide specific syntax for operating system calls, and most assembly languages can be used universally with any operating system, as the language provides access to all the real capabilities of the processor, upon which all system call mechanisms ultimately rest. In contrast to assembly languages, most high-level programming languages are generally portable across multiple architectures but require interpreting or compiling, a much more complicated task than assembling.

The computational step when an assembler is processing a program is called*assembly time*.

Assembly language syntax

Assembly language uses a mnemonic to represent each low-level machine instruction or opcode, typically also each architectural register, flag, etc. Many operations require one or more operands in order to form a complete instruction. Most assemblers permit named constants, registers, and labels for program and memory locations, and can calculate expressions for operands. Thus, programmers are freed from tedious repetitive calculations and assembler programs are much more readable than machine code. Depending on the architecture, these elements may also be combined for specific instructions or addressing modes using offsets or other data as well as fixed addresses. Many assemblers offer additional mechanisms to facilitate program development, to control the assembly process, and to aid debugging.

Terminology

- A **macro assembler** is an assembler that includes a macroinstruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code.

- A **cross assembler** (see also cross compiler) is an assembler that is run on a computer or operating system (the *host* system) of a different type from the system on which the resulting code is to run (the *target system*). Cross-assembling facilitates the development of programs for systems that do not have the resources to support software development, such as an embedded system or a microcontroller. In such a case, the resulting object code must be transferred to the target system, via read-only memory (ROM, EPROM, etc.), a programmer (when the read-only memory is integrated in the device, as in microcontrollers), or a data link using either an exact bit-by-bit copy of the object code or a text-based representation of that code (such as Intel hex or Motorola S-record).
- A **high-level assembler** is a program that provides language abstractions more often associated with high-level languages, such as advanced control structures (IF/THEN/ELSE, DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.
- A **microassembler** is a program that helps prepare a microprogram, called *firmware*, to control the low level operation of a computer.
- A **meta-assembler** is "a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language". "Meta-Symbol" assemblers for the SDS 9 Series and SDS Sigma series of computers are meta-

assemblers. Sperry Univac also provided a Meta-Assembler for the UNIVAC 1100/2200 series.

- **inline assembler** (or **embedded assembler**) is assembler code contained within a high-level language program. This is most often used in systems programs which need direct access to the hardware.

Key concepts

Assembler

An **assembler** program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. This representation typically includes an *operation code* ("opcode") as well as other control bits and data. The assembler also calculates constant expressions and resolves 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.

Some assemblers may also be able to perform some simple types of instruction set-specific optimizations. One concrete example of this may be the ubiquitous x86 assemblers from various vendors. Called jump-sizing, most of them are able to perform jump-instruction replacements (long jumps replaced by short or relative jumps) in any number of passes, on request. Others may even do simple rearrangement or insertion of instructions, such as some assemblers for RISCarchitectures that can help optimize a sensible instruction scheduling to exploit the CPU pipeline as efficiently as possible.

Assemblers have been available since the 1950s, as the first step above machine language and before high-level programming languages such as Fortran, Algol, COBOL and Lisp. There have also been several classes of translators and semi-automatic code generators with properties similar to both assembly and high-level languages, with Speedcode as perhaps one of the better-known examples.

There may be several assemblers with different syntax for a particular CPU or instruction set architecture. For instance, an instruction to add memory data to a register in a x86-family processor might be `add eax,[ebx]`, in original *Intel syntax*, whereas this would be written `addl (%ebx),%eax` in the *AT&T syntax* used by the GNU Assembler. Despite different appearances, different syntactic forms generally generate the same numeric machine code. A single assembler may also have different modes in order to support variations in syntactic forms as well as their exact semantic interpretations (such as FASM-syntax, TASM-syntax, ideal mode, etc., in the special case of x86 assembly programming).

Number of passes

There are two types of assemblers based on how many passes through the source are needed (how many times the assembler reads the source) to produce the object file.

- **One-pass assemblers** go through the source code once. Any symbol used before it is defined will require "errata" at the end of the object code (or, at least, no earlier than the point where the symbol is defined) telling the linker or the loader to "go back" and overwrite a placeholder which had been left where the as yet undefined symbol was used.
- **Multi-pass assemblers** create a table with all symbols and their values in the first passes, then use the table in later passes to generate code.

In both cases, the assembler must be able to determine the size of each instruction on the initial passes in order to calculate the addresses of subsequent symbols. This means that if the size of an operation referring to an operand defined later depends on the type or distance of the operand, the assembler will make a pessimistic estimate when first encountering the operation, and if necessary, pad it with one or more "no-operation" instructions in a later pass or the errata. In an assembler with peephole optimization, addresses may be recalculated between passes to allow replacing pessimistic code with code tailored to the exact distance from the target.

The original reason for the use of one-pass assemblers was memory size and speed of assembly – often a second pass would require storing the symbol table in memory (to handle forward references), rewinding and rereading the program source on tape, or rereading a deck of cards or punched paper tape. Later computers with much larger memories (especially disc storage), had the space to perform all necessary processing without such re-reading. The advantage of the multi-pass assembler is that the absence of errata makes the linking process (or the program load if the assembler directly produces executable code) faster.

Example: in the following code snippet, a one-pass assembler would be able to determine the address of the backward reference *BKWD* when assembling statement *S2*, but would not be able to determine the address of the forward reference *FWD* when assembling the branch statement *S1*; indeed, *FWD* may be undefined. A two-pass assembler would determine both addresses in pass 1, so they would be known when generating code in pass 2.

```
S1   B   FWD
  ...
FWD  EQU *
  ...
BKWD EQU *
  ...
S2   B   BKWD
```

High-level assemblers

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

- High-level procedure/function declarations and invocations
- Advanced control structures (IF/THEN/ELSE, SWITCH)
- High-level abstract data types, including structures/records, unions, classes, and sets
- Sophisticated macro processing (although available on ordinary assemblers since the late 1950s for, e.g., the IBM 700 series and IBM 7000 series, and since

the 1960s for IBM System/360 (S/360), amongst other machines)

- Object-oriented programming features such as classes, objects, abstraction, polymorphism, and inheritance

See Language design below for more details.

Assembly language

A program written in assembly language consists of a series of mnemonic processor instructions and meta-statements (known variously as directives, pseudo-instructions, and pseudo-ops), comments and data. Assembly language instructions usually consist of an opcode mnemonic followed by an operand, which might be a list of data, arguments or parameters. Some instructions may be "implied," which means the data upon which the instruction operates is implicitly defined by the instruction itself—such an instruction does not take an operand. The resulting statement is translated by an assembler into machine language instructions that can be loaded into memory and executed.

For example, the instruction below 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. Assembly language for the 8086 family provides the mnemonicMOV (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.

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

In some assembly languages (including this one) the same mnemonic, such as MOV, may be used for a family of related instructions for loading, copying and moving data, whether these are immediate values, values in registers, or memory locations pointed to by values in registers or by immediate (a.k.a direct) addresses. Other assemblers may use separate opcode mnemonics such as L for "move memory to register", ST for "move register to memory", LR for "move register to register", MVI for "move immediate operand to memory", etc.

If the same mnemonic is used for different instructions, that means that the mnemonic corresponds to several different binary instruction codes, excluding data (e.g. the `61h` in this example), depending on the operands that follow the mnemonic. For example, for the x86/IA-32 CPUs, the Intel assembly language syntax `MOV AL, AH` represents an instruction that moves the contents of register *AH* into register *AL*. The hexadecimal form of this instruction is:

88 E0

The first byte, 88h, identifies a move between a byte-sized register and either another register or memory, and the second byte, E0h, is encoded (with three bit-fields) to specify that both operands are registers, the source is *AH*, and the destination is *AL*.

In a case like this where the same mnemonic can represent more than one binary instruction, the assembler determines which instruction to generate by examining the operands. In the first example, the operand `61h` is a valid hexadecimal numeric constant and is not a valid register name, so only the `B0` instruction can be applicable. In the second example, the operand `AH` is a valid register name and not a valid numeric constant (hexadecimal, decimal, octal, or binary), so only the `88` instruction can be applicable.

Assembly languages are always designed so that this sort of unambiguousness is universally enforced by their syntax. For example, in the Intel x86 assembly language, a hexadecimal constant must start with a numeral digit, so that the hexadecimal number 'A' (equal to decimal ten) would be written as `0Ah` or `0AH`, not `AH`, specifically so that it cannot appear to be the name of register *AH*. (The same rule also prevents ambiguity with the names of registers *BH*, *CH*, and *DH*, as well as with any user-defined symbol that ends with the letter *H* and otherwise contains only characters that are hexadecimal digits, such as the word "BEACH".)

Returning to the original example, while the x86 opcode 10110000 (`B0`) copies an 8-bit value into the *AL* register, 10110001 (B1) moves it into *CL*and 10110010 (B2) does so into *DL*. Assembly language examples for these follow.

```
MOVAL,1h; Load AL with immediate value 1
MOVCL,2h; Load CL with immediate value 2
MOVDL,3h; Load DL with immediate value 3
```

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

```
MOVEAX,[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
MOVDS,DX; Move the contents of DX into segment register DS
```

In each case, the MOV mnemonic is translated directly into one of the opcodes 88-8C, 8E, A0-A3, B0-BF, C6 or C7 by an assembler, and the programmer normally 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 a one-to-one correspondence between many 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. Since the information about pseudoinstructions and macros defined in the assembler environment is not present in the object program, a disassembler cannot reconstruct the macro and pseudoinstruction invocations but can only disassemble the actual machine instructions that the assembler generated from those abstract assembly-language entities. Likewise, since comments in the assembly language source file are ignored by the assembler and have no effect on the object code it generates, a disassembler is always completely unable to recover source comments.

Each computer architecture has its own machine language. Computers differ in the number and type of operations they support, in the different sizes and numbers of registers, and in the representations of data 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 CPU manufacturer and used in its documentation.

Two examples of CPUs that have two different sets of mnemonics are the Intel 8080 family and the Intel 8086/8088. Because Intel claimed copyright on its assembly language mnemonics (on each page of their documentation published in the 1970s and early 1980s, at least), some companies that independently produced CPUs compatible with Intel instruction sets invented their own mnemonics. The Zilog Z80 CPU, an enhancement of the Intel 8080A, supports all the 8080A instructions plus many more; Zilog invented an entirely new assembly language, not only for the new instructions but also for all of the 8080A instructions. For example, where Intel uses the mnemonics *MOV*, *MVI*, *LDA*, *STA*, *LXI*, *LDAX*, *STAX*, *LHLD*, and *SHLD* for various data transfer instructions, the Z80 assembly language uses the mnemonic *LD* for all of them. A similar case is the NEC V20 and V30 CPUs, enhanced copies of the Intel 8086 and 8088, respectively. Like Zilog with the Z80, NEC invented new mnemonics for all of the 8086 and 8088 instructions, to avoid accusations of infringement of Intel's copyright. (It is questionable whether such copyrights can be valid, and later CPU companies such as AMD and Cyrix republished Intel's x86/IA-32 instruction mnemonics exactly with neither permission nor legal penalty.) It is doubtful whether in practice many people who programmed the V20 and V30 actually wrote in NEC's assembly language rather than Intel's; since any two assembly languages for the same instruction set architecture are isomorphic (somewhat like English and Pig Latin), there is no requirement to use a manufacturer's own published assembly language with that manufacturer's products.

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 definitions
- Assembly directives

Opcode mnemonics and extended mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in high-level languages. 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 (value coded in the instruction itself), registers specified in the instruction or 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` ("NO OPeration" – do nothing for one step) 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 xchgax,ax. Some disassemblers recognize this and will decode the xchgax,ax instruction as nop. Similarly, IBM assemblers for System/360 and System/370 use the extended mnemonics NOP and NOPR for BC and BCR with zero masks. For the SPARC architecture, these are known as *synthetic instructions*.

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

Mnemonics are arbitrary symbols; in 1985 the IEEE published Standard 694 for a uniform set of mnemonics to be used by all assemblers. The standard has since been withdrawn.

Data directives

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 commands given to an assembler "directing it to perform operations other than assembling instructions". Directives affect how the assembler operates and "may affect the object code, the symbol table, the listing file, and the values of internal assembler parameters". Sometimes the term *pseudo-opcode*is reserved for directives that generate object code, such as those that generate data.

The names of pseudo-ops often start with a dot to distinguish them from machine instructions. Pseudo-ops can make the assembly of the program dependent on parameters input by a programmer, so that one program can be assembled in different ways, perhaps for different applications. Or, a pseudo-op can be used to manipulate presentation of a program to make it easier to read and maintain. Another common use of pseudo-ops is to reserve storage areas for run-time data and optionally initialize their contents to known values.

Symbolic assemblers let programmers associate arbitrary names (*labels* or *symbols*) with memory locations and various constants. 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 often lexically distinct from normal symbols (e.g., the use of "10$" as a GOTO destination).

Some assemblers, such as NASM, 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 program source code that will be ignored during assembly. Judicious commenting is essential in assembly language programs, as the meaning and purpose of a sequence of binary machine instructions can be difficult to determine. The "raw" (uncommented) assembly language generated by compilers or disassemblers 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. The macro definition is most commonly a mixture of assembler statements, e.g., directives, symbolic machine instructions, and templates for assembler statements. 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). Macros in this sense date to IBM autocoders of the 1950s.

In assembly language, the term "macro" represents a more comprehensive concept than it does in some other contexts, such as the pre-processor in the C programming language, where its #define directive typically is used to create short single line macros. Assembler macro instructions, like macros in PL/I and some other languages, can be lengthy "programs" by themselves, executed by interpretation by the assembler during assembly.

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.

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 numerous 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. For instance, a "sort" macro could accept the specification of a complex sort key and generate code crafted for that specific key, not needing the run-time tests that would be required for a general procedure interpreting the specification. 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. Underlining this point, macros were used to implement an early virtual machine in SNOBOL4 (1967), which was written in the SNOBOL Implementation Language (SIL), an assembly language for a virtual machine. The target machine would translate this to its native code using a macro assembler. This allowed a high degree of portability for the time.

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 (VM/CMS) and with IBM's "real time transaction processing" add-ons, Customer Information Control System CICS, and ACP/TPF, the airline/financial system that began in the 1970s and still runs many large computer reservation systems (CRS) and credit card systems today.

It is 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. IBM OS/360 uses macros to perform system generation. The user specifies options by coding a series of assembler macros. Assembling these macros generates a job stream to build the system, including job control language and utility control statements.

This is 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. Unlike certain previous macro processors inside assemblers, the C preprocessor is not Turing-complete because it lacks 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 (major exceptions being C, C++ and PL/I) while remaining a perennial for assemblers.

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.

Support for structured programming

Packages of macros have been written providing structured programming elements to encode execution flow. The earliest example of this approach was in the Concept-14 macro set, originally proposed by Harlan Mills (March 1970), and implemented by Marvin Kessler at IBM's Federal Systems Division, which provided IF/ELSE/ENDIF and similar control flow blocks for OS/360 assembler programs. 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 1980s (the latter days of large-scale assembly language use). IBM's High Level Assembler Toolkit includes such a macro package.

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 Ccompiler). 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.

Assemblers with a strong macro engine allow structured programming via macros, such as the switch macro provided with the Masm32 package (this code is a complete program):

```
include\masm32\include\masm32rt.inc        ; use the Masm32 library

.code
demomain:
REPEAT20
          switchrv(nrandom,9)          ; generate a number between 0 and 8
          movecx,7
          case0
                    print"case 0"
          caseecx                                ; in contrast to most other programming
languages,
                    print"case 7"              ; the Masm32 switch allows "variable cases"
          case1..3
                    .ifeax==1
                              print"case 1"
                    .elseifeax==2
                              print"case 2"
                    .else
                              print"cases 1 to 3: other"
                    .endif
          case4,6,8
                    print"cases 4, 6 or 8"
          default
                    movebx,19                  ; print 20 stars
                    .Repeat
                              print"*"
                              decebx
                    .UntilSign?                ; loop until the sign flag is set
          endsw
          printchr$(13,10)
ENDM
exit
enddemomain
```

Use of assembly language

Historical perspective

Assembly languages were not available at the time when the stored-program computerwas introduced. Kathleen Booth "is credited with inventing assembly language" based on theoretical work she began in 1947, while working on the ARC2 at Birkbeck, University of London following consultation by Andrew Booth (later her husband) with mathematician John von Neumann and physicist Herman Goldstine at the Institute for Advanced Study.

In late 1948, the Electronic Delay Storage Automatic Calculator (EDSAC) had an assembler (named "initial orders") integrated into its bootstrap program. It used one-letter mnemonics developed by David Wheeler, who is credited by the IEEE Computer Society as the creator of the first "assembler". Reports on the EDSAC introduced the term "assembly" for the process of combining fields into an instruction word. SOAP (Symbolic Optimal Assembly Program) was an assembly language for the IBM 650 computer written by Stan Poley in 1955.

Assembly languages eliminate much of the error-prone, tedious, and time-consuming first-generation programming needed with the earliest computers, freeing programmers from tedium such as remembering numeric codes and calculating addresses.

Assembly languages were once widely used for all sorts of programming. However, by the 1980s (1990s on microcomputers), their use had largely been supplanted by higher-level languages, in the search for improved programming productivity. Today, assembly language 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, numerous programs have been written entirely in assembly language. The Burroughs MCP (1961) was the first computer for which an operating system was not developed entirely in assembly language; it was written in Executive Systems Problem Oriented Language (ESPOL), an Algol dialect. Many commercial applications were written in assembly language as well, including a large amount of the IBM mainframe software written by large corporations. COBOL, FORTRAN and some PL/I eventually displaced much of this work, although a number of large organizations retained assembly-language application infrastructures well into the 1990s.

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, the Turbo Pascal compiler and early applications such as the spreadsheet program Lotus 1-2-3. Assembly language was used to get the best performance out of the Sega Saturn, a console that was notoriously challenging to develop and program games for. The 1993 arcade game *NBA Jam* is another example.

Assembly language has long been the primary development language for many popular home computers of the 1980s and 1990s (such as the MSX, SinclairZX Spectrum, Commodore 64, Commodore Amiga, and Atari ST). This was in large part because interpreted 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 even have an integrated development environment (IDE) with highly advanced debugging and macro facilities. Some compilers available for the Radio ShackTRS-80 and its successors had the capability to combine inline assembly source with high-level program statements. Upon compilation, a built-in assembler produced inline machine code.

Current usage

There have always been debates over the usefulness and performance of assembly language relative to high-level languages.

Although assembly language has specific niche uses where it is important (see below), there are other tools for optimization.

As of July 2017, the TIOBE index of programming language popularity ranks assembly language at 11, ahead of Visual Basic, for example. Assembler can be used to optimize for speed or optimize for size. In the case of speed optimization, modern optimizing compilersare 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-systems makes effective optimization increasingly difficult for compilers, as well as for assembly programmers. Moreover, increasing processor performance has meant that most CPUs sit idle most of the time, with delays caused by predictable bottlenecks such as cache misses, I/O operations and paging. This has made raw code execution speed a non-issue for many programmers.

There are some situations in which developers might choose to use assembly language:

- Writing code for systems with older processors that have limited high-level language options such as the Atari 2600, Commodore 64, and graphing calculators. Programs for these computers of 1970s and 1980s are often written in the context of demoscene or retrogaming subcultures.
- Code that must interact directly with the hardware, for example in device drivers and interrupt handlers.
- In an embedded processor or DSP, high-repetition interrupts require the shortest number of cycles per interrupt, such as an interrupt that occurs 1000 or 10000 times a second.
- Programs that need to use processor-specific instructions not implemented in a compiler. A common example is the bitwise rotation instruction at the core of many encryption algorithms, as well as

querying the parity of a byte or the 4-bit carry of an addition.

- A stand-alone executable of compact size is required that must execute without recourse to the run-time components or libraries associated with a high-level language. Examples have included firmware for telephones, automobile fuel and ignition systems, air-conditioning control systems, security systems, and sensors.
- Programs with performance-sensitive inner loops, where assembly language provides optimization opportunities that are difficult to achieve in a high-level language. For example, linear algebra with BLAS or discrete cosine transformation (e.g. SIMD assembly version from x264).
- Programs that create 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.
- Real-time programs such as simulations, flight navigation systems, and medical equipment. For example, in a fly-by-wiresystem, 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.

- Cryptographic algorithms that must always take strictly the same time to execute, preventing timing attacks.
- Modify and extend legacy code written for IBM mainframe computers.
- Situations where complete control over the environment is required, in extremely high-security situations where nothing can be taken for granted.
- Computer viruses, bootloaders, certain device drivers, or other items very close to the hardware or low-level operating system.
- Instruction set simulators for monitoring, tracing and debugging where additional overhead is kept to a minimum.
- Situations where no high-level language exists, on a new or specialized processor for which no cross compiler is available.
- Reverse-engineering and modifying program files such as:
- existingbinaries that may or may not have originally been written in a high-level language, for example when trying to recreate programs for which source code is not available or has been lost, or cracking copy protection of proprietary software.
- Video games (also termed ROM hacking), which is possible via several methods. The most widely employed method is altering program code at the assembly language level.

Assembly language is still taught in most computer science and electronic engineering programs. Although few programmers today regularly work with assembly language as a tool, the underlying concepts remain 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

- Assembly language is typically used in a system's boot code, the low-level code that initializes and tests the system hardware prior to booting the operating system and is often stored in ROM. (BIOS on IBM-compatible PC systems and CP/M is an example.)
- Assembly language is often used for low-level code, for instance for operating system kernels, which cannot rely on the availability of pre-existing system calls and must indeed implement them for the particular processor architecture on which the system will be running.

- Some compilers translate high-level languages into assembly first before fully compiling, allowing the assembly code to be viewed for debugging and optimization purposes.
- Some compilers for relatively low-level languages, such as Pascal or C, allow the programmer to embed assembly language directly in the source code (so called inline assembly). Programs using such facilities 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 useful in reverse engineering. Many programs are distributed only in machine code form which is straightforward to translate into assembly language by a disassembler, but more difficult to translate into a higher-level language through a decompiler. Tools such as the Interactive Disassembler make extensive use of disassembly for such a purpose. This technique is used by hackers to crack commercial software, and competitors to produce software with similar results from competing companies.
- Assembly language is used to enhance speed of execution, especially in early personal computers with limited processing power and RAM.
- Assemblers can be used to generate blocks of data, with no high-level language overhead, from formatted and commented source code, to be used by other code.