

Programming-1001 Wiki Collection

--Programming Paradigms & language implementations.

Contents

Articles

Intro	1
Programming language	1
Programming paradigm	15
Prehistoric	20
Procedural programming	20
Structured programming	23
Fortran	27
From Math to Programming	45
Lambda calculus	45
Closure (computer science)	61
Functional programming	74
Lisp (programming language)	85
Pattern matching	107
Anonymous function	113
Scheme (programming language)	135
Closer to Math	154
Type inference	154
Currying	157
ML (programming language)	160
Standard ML	163
Tail call	176
Lazy evaluation	184
Haskell (programming language)	189
Type system	198
Scope (computer science)	211
Masterpiece of Alan Kay	218
Class (computer programming)	218
Object (computer science)	228
Object-oriented programming	231
Smalltalk	242

Common Lisp	256
OCaml	279
Prototype	288
Prototype-based programming	288
Self (programming language)	293
JavaScript	299
The code that writes code	318
Metaprogramming	318
Generic programming	321
Ada (programming language)	338
C++	351
That's a duck	364
Duck typing	364
Ruby (programming language)	377
Be more, be complex...	393
Concurrency (computer science)	393
Erlang (programming language)	396
New Guys	406
D (programming language)	406
Go (programming language)	415
References	
Article Sources and Contributors	421
Image Sources, Licenses and Contributors	429
Article Licenses	
License	430

Intro

Programming language

A **programming language** is an artificial language designed to communicate instructions to a machine, particularly a computer. Programming languages can be used to create programs that control the behavior of a machine and/or to express algorithms precisely.

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

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 5 and earlier, have a dominant implementation that is used as a reference.

Definitions

A programming language is a notation for writing programs, which are specifications of a computation or algorithm.^[1] Some, but not all, authors restrict the term "programming language" to those languages that can express *all* possible algorithms.^{[1][2]} Traits often considered important for what constitutes a programming language include:

- *Function and target:* A *computer programming language* is a language^[3] used to write computer programs, which involve a computer performing some kind of computation^[4] or algorithm and possibly control external devices such as printers, disk drives, robots,^[5] 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.^[6] In most practical contexts, a programming language involves a computer; consequently, programming languages are usually defined and studied this way.^[7] 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;^[8] this principle is sometimes formulated as recommendation to the programmer to make proper use of such abstractions.^[9]
- *Expressive power:* The theory of computation classifies languages by the computations they are capable of expressing. All Turing complete languages can implement the same set of algorithms. ANSI/ISO SQL and Charity are examples of languages that are not Turing complete, yet often called programming languages.^{[10][11]}

Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages.^{[12][13][14]} Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect.^{[15][16][17]} Moreover, LaTeX, which is mostly used for structuring documents, also contains a Turing complete subset.^{[18][19]}

The term *computer language* is sometimes used interchangeably with programming language.^[20] 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.^[21] In this vein, languages used in computing that have a different goal than expressing computer programs are generically designated computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming.^[22] 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.^[23] 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.^[24]

Elements

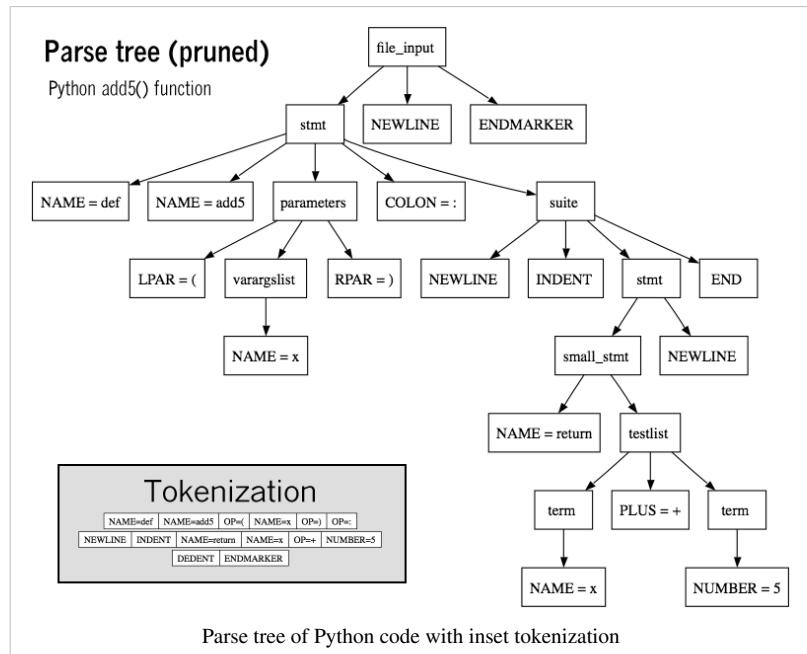
All programming languages have some primitive building blocks for the description of data and the processes or transformations applied to them (like the addition of two numbers or the selection of an item from a collection). These primitives are defined by syntactic and semantic rules which describe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

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

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

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

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

This grammar specifies the following:

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

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

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

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

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

The following C language fragment is syntactically correct, but performs 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;
complex abs_p = sqrt(*p >> 4 + p->im);
```

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

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars.^[25] 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.^[26] 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.^[27]

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.^[1] 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.^[28] 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, with the implication that it is not applicable to other types.^[29] For example, the data represented by "this text between the quotes" is a string. In most programming languages, dividing a number by a string has no meaning. Most modern programming languages will therefore reject any program attempting to perform such an operation. In some languages, the meaningless operation will be detected when the program is compiled ("static" type checking), and rejected by the compiler, while in others, it will be detected when the program is run ("dynamic" type checking), resulting in a runtime exception.

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

In contrast, an *untyped language*, such as most assembly languages, allows any operation to be performed on any data, which are generally considered to be sequences of bits of various lengths.^[29] High-level languages which are untyped include BCPL and some varieties of Forth.

In practice, while few languages are considered typed from the point of view of type theory (verifying or rejecting *all* operations), most modern languages offer a degree of typing.^[29] Many production languages provide means to bypass or subvert the type system (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.^[29]

Statically typed languages can be either *manifestly typed* or *type-inferred*. In the first case, the programmer must explicitly write types at certain textual positions (for example, at variable declarations). In the second case, the compiler *infers* the types of expressions and declarations based on context. Most mainstream statically typed languages, such as C++, C# and Java, are manifestly typed. Complete type inference has traditionally been associated with less mainstream languages, such as Haskell and ML. However, many manifestly typed languages support partial type inference; for example, Java and C# both infer types in certain limited cases.^[30]

Dynamic typing, also called *latent typing*, determines the type-safety of operations at runtime; in other words, types are associated with *runtime values* rather than *textual expressions*.^[29] 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, Perl, Python, JavaScript, and Ruby are dynamically typed.

Weak and strong typing

Weak typing allows a value of one type to be treated as another, for example treating a string as a number.^[29] 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 the above. An attempt to perform an operation on the wrong type of value raises an error.^[29] 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.^{[31][32]}

Standard library and run-time system

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

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

Design and implementation

Programming languages share properties with natural languages related to their purpose as vehicles for communication, having a syntactic form separate from its semantics, and showing *language families* of related languages branching one from another.^{[3][33]} 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.^[34] 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.^[35] The need for diverse programming languages arises from the diversity of contexts in which languages are used:

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

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

less effort from the programmer. This lets them write more functionality per time unit.^[36]

Natural language processors have been proposed as a way to eliminate the need for a specialized language for programming. However, this goal remains distant and its benefits are open to debate. Edsger W. Dijkstra took the position that the use of a formal language is essential to prevent the introduction of meaningless constructs, and dismissed natural language programming as "foolish".^[37] Alan Perlis was similarly dismissive of the idea.^[38] Hybrid approaches have been taken in Structured English and SQL.

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

Specification

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

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

- An explicit definition of the syntax, static semantics, and execution semantics of the language. While syntax is commonly specified using a formal grammar, semantic definitions may be written in natural language (e.g., as in the C language), or a formal semantics (e.g., as in Standard ML^[39] and Scheme^[40] 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^[41]). The syntax and semantics of the language are explicit in the behavior of the reference implementation.

Implementation

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

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

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

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

Usage

Thousands of different programming languages have been created, mainly in the computing field.^[42] 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).^[43] *Programming* is the process by which programmers combine these primitives to compose new programs, or adapt existing ones to new uses or a changing environment.

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

Measuring language usage

It is difficult to determine which programming languages are most widely used, and what usage means varies by context. One language may occupy the greater number of programmer hours, a different one have more lines of code, and a third utilize the most CPU time. Some languages are very popular for particular kinds of applications. For example, COBOL is still strong in the corporate data center, often on large mainframes; Fortran in scientific and engineering applications; 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^[44]
- the number of books sold that teach or describe the language^[45]
- estimates of the number of existing lines of code written in the language—which may underestimate languages not often found in public searches^[46]
- counts of language references (i.e., to the name of the language) found using a web search engine.

Combining and averaging information from various internet sites, langpop.com claims that^[47] in 2008 the 10 most cited programming languages are (in alphabetical order): C, C++, C#, Java, JavaScript, Perl, PHP, Python, Ruby, and SQL.

Taxonomies

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

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

In broad strokes, programming languages divide into *programming paradigms* and a classification by *intended domain of use*. Traditionally, programming languages have been regarded as describing computation in terms of imperative sentences, i.e. issuing commands. These are generally called imperative programming languages. A great deal of research in programming languages has been aimed at blurring the distinction between a program as a set of instructions and a program as an assertion about the desired answer, which is the main feature of declarative programming.^[48] 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).^[49] Some general purpose languages were designed largely with educational goals.^[50]

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.

History

Early developments

The first programming languages predate the modern computer. The 19th century saw the invention of "programmable" looms and player piano scrolls, both of which implemented examples of domain-specific languages. By the beginning of the twentieth century, punch cards encoded data and directed mechanical processing. In the 1930s and 1940s, the formalisms of Alonzo Church's lambda calculus and Alan Turing's Turing machines provided mathematical abstractions for expressing algorithms; the lambda calculus remains influential in language design.^[51]

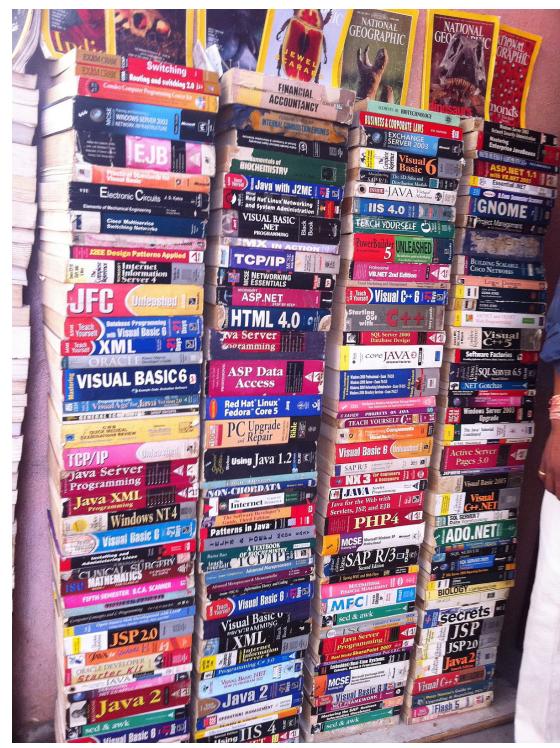
In the 1940s, the first electrically powered digital computers were created. Grace Hopper, was one of the first programmers of the Harvard Mark I computer, a pioneer in the field, developed the first compiler, around 1952, for a computer programming language. Notwithstanding, the idea of programming language existed earlier; the first high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.^[52]

Programmers of early 1950s computers, notably UNIVAC I and IBM 701, used machine language programs, that is, the first generation language (1GL). 1GL programming was quickly superseded by similarly machine-specific, but mnemonic, second generation languages (2GL) known as assembly languages or "assembler". Later in the 1950s, assembly language programming, which had evolved to include the use of macro instructions, was followed by the development of "third generation" programming languages (3GL), such as FORTRAN, LISP, and COBOL.^[53] 3GLs are more abstract and are "portable", or at least implemented similarly on computers that do not support the same native machine code. Updated versions of all of these 3GLs are still in general use, and each has strongly influenced the development of later languages.^[54] At the end of the 1950s, the language formalized as ALGOL 60 was introduced, and most later programming languages are, in many respects, descendants of Algol.^[54] The format and use of the early programming languages was heavily influenced by the constraints of the interface.^[55]

Refinement

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

- APL introduced *array programming* and influenced functional programming.^[56]
- PL/I (NPL) was designed in the early 1960s to incorporate the best ideas from FORTRAN and COBOL.
- In the 1960s, Simula was the first language designed to support *object-oriented programming*; in the mid-1970s, Smalltalk followed with the first "purely" object-oriented language.
- C was developed between 1969 and 1973 as a *system programming* language, and remains popular.^[57]
- Prolog, designed in 1972, was the first *logic programming* language.



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

- In 1978, ML built a polymorphic type system on top of Lisp, pioneering *statically typed functional programming* languages.

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

The 1960s and 1970s also saw considerable debate over the merits of *structured programming*, and whether programming languages should be designed to support it.^[58] Edsger Dijkstra, in a famous 1968 letter published in the Communications of the ACM, argued that GOTO statements should be eliminated from all "higher level" programming languages.^[59]

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

Consolidation and growth

The 1980s were years of relative consolidation. C++ combined object-oriented and systems programming. The United States government standardized Ada, a systems programming language derived from Pascal and intended for use by defense contractors. In Japan and elsewhere, vast sums were spent investigating so-called "fifth generation" languages that incorporated logic programming constructs.^[60] The functional languages community moved to standardize ML and Lisp. Rather than inventing new paradigms, all of these movements elaborated upon the ideas invented in the previous decade.

One important trend in language design for programming large-scale systems during the 1980s was an increased focus on the use of *modules*, or large-scale organizational units of code. Modula-2, Ada, and ML all developed notable module systems in the 1980s, although other languages, such as PL/I, already had extensive support for modular programming. Module systems were often wedded to generic programming constructs.^[61]

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

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

The 4GLs are examples of languages which are domain-specific, such as SQL, which manipulates and returns sets of data rather than the scalar values which are canonical to most programming languages. Perl, for example, with its 'here document' can hold multiple 4GL programs, as well as multiple JavaScript programs, in part of its own perl code and use variable interpolation in the 'here document' to support multi-language programming.^[62]

References

- [1] Aaby, Anthony (2004). *Introduction to Programming Languages* (<http://www.emu.edu.tr/aelci/Courses/D-318/D-318-Files/plbook/intro.htm>) .
- [2] In mathematical terms, this means the programming language is Turing-complete MacLennan, Bruce J. (1987). *Principles of Programming Languages*. Oxford University Press. p. 1. ISBN 0-19-511306-3.
- [3] Steven R. Fischer, *A history of language*, Reaktion Books, 2003, ISBN 1-86189-080-X, p. 205
- [4] ACM SIGPLAN (2003). "Bylaws of the Special Interest Group on Programming Languages of the Association for Computing Machinery" (http://www.acm.org/sigs/sigplan/sigplan_bylaws.htm). . Retrieved 19 June 2006., *The scope of SIGPLAN is the theory, design, implementation, description, and application of computer programming languages - languages that permit the specification of a variety of different computations, thereby providing the user with significant control (immediate or delayed) over the computer's operation.*
- [5] Dean, Tom (2002). "Programming Robots" (<http://www.cs.brown.edu/people/tld/courses/cs148/02/programming.html>). *Building Intelligent Robots*. Brown University Department of Computer Science. . Retrieved 23 September 2006.
- [6] R. Narasimhan, Programming Languages and Computers: A Unified Metatheory, pp. 189--247 in Franz Alt, Morris Rubinoff (eds.) Advances in computers, Volume 8, Academic Press, 1994, ISBN 012012108, p.193 : "a complete specification of a programming language must, by definition, include a specification of a processor--idealized, if you will--for that language." [the source cites many references to support this statement]
- [7] Ben Ari, Mordechai (1996). *Understanding Programming Languages*. John Wiley and Sons. "Programs and languages can be defined as purely formal mathematical objects. However, more people are interested in programs than in other mathematical objects such as groups, precisely because it is possible to use the program—the sequence of symbols—to control the execution of a computer. While we highly recommend the study of the theory of programming, this text will generally limit itself to the study of programs as they are executed on a computer."
- [8] David A. Schmidt, *The structure of typed programming languages*, MIT Press, 1994, ISBN 0-262-19349-3, p. 32
- [9] Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. p. 339. ISBN 0-262-16209-1.
- [10] Digital Equipment Corporation. "Information Technology - Database Language SQL (Proposed revised text of DIS 9075)" (<http://www.contrib.andrew.cmu.edu/~shadow/sql/sql1992.txt>). *ISO/IEC 9075:1992, Database Language SQL* . Retrieved 29 June 2006.
- [11] The Charity Development Group (December 1996). "The CHARITY Home Page" (<http://pli.cpsc.ucalgary.ca/charity1/www/home.html>). . Retrieved 29 June 2006., *Charity is a categorical programming language..., All Charity computations terminate.*
- [12] XML in 10 points (<http://www.w3.org/XML/1999/XML-in-10-points.html>) W3C, 1999, *XML is not a programming language*.
- [13] Powell, Thomas (2003). *HTML & XHTML: the complete reference*. McGraw-Hill. p. 25. ISBN 0-07-222942-X. *"HTML is not a programming language."*
- [14] Dykes, Lucinda; Tittel, Ed (2005). *XML For Dummies, 4th Edition*. Wiley. p. 20. ISBN 0-7645-8845-1. *"...it's a markup language, not a programming language."*
- [15] "What kind of language is XSLT?" (<http://www.ibm.com/developerworks/library/x-xslt/>). Ibm.com. . Retrieved 3 December 2010.
- [16] "XSLT is a Programming Language" ([http://msdn.microsoft.com/en-us/library/ms767587\(VS.85\).aspx](http://msdn.microsoft.com/en-us/library/ms767587(VS.85).aspx)). Msdn.microsoft.com. . Retrieved 3 December 2010.
- [17] Scott, Michael (2006). *Programming Language Pragmatics*. Morgan Kaufmann. p. 802. ISBN 0-12-633951-1. *"XSLT, though highly specialized to the transformation of XML, is a Turing-complete programming language."*
- [18] <http://tobi.oetiker.ch/lshort/lshort.pdf>
- [19] Syropoulos, Apostolos; Antonis Tsolomitis, Nick Sofroniou (2003). *Digital typography using LaTeX*. Springer-Verlag. p. 213. ISBN 0-387-95217-9. *"TeX is not only an excellent typesetting engine but also a real programming language."*
- [20] Robert A. Edmunds, The Prentice-Hall standard glossary of computer terminology, Prentice-Hall, 1985, p. 91
- [21] Pascal Lando, Anne Lapujade, Gilles Kassel, and Frédéric Fürst, *Towards a General Ontology of Computer Programs* (http://www.loa-cnrs.fr/ICSOFT2007_final.pdf), ICSOFT 2007 (<http://dblp.uni-trier.de/db/conf/icsoft/icsoft2007-1.html>), pp. 163-170
- [22] S.K. Bajpai, *Introduction To Computers And C Programming*, New Age International, 2007, ISBN 81-224-1379-X, p. 346
- [23] R. Narasimhan, Programming Languages and Computers: A Unified Metatheory, pp. 189--247 in Franz Alt, Morris Rubinoff (eds.) Advances in computers, Volume 8, Academic Press, 1994, ISBN 012012108, p.215: "[...] the model [...] for computer languages differs from that [...] for programming languages in only two respects. In a computer language, there are only finitely many names--or registers--which can assume only finitely many values--or states--and these states are not further distinguished in terms of any other attributes. [author's footnote:] This may sound like a truism but its implications are far reaching. For example, it would imply that any model for programming languages, by fixing certain of its parameters or features, should be reducible in a natural way to a model for computer languages."
- [24] John C. Reynolds, *Some thoughts on teaching programming and programming languages*, SIGPLAN Notices, Volume 43, Issue 11, November 2008, p.109
- [25] Michael Sipser (1996). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X. Section 2.2: Pushdown Automata, pp.101–114.
- [26] Jeffrey Kegler, " Perl and Undecidability (<http://www.jeffreykegler.com/Home/perl-and-undecidability>)", *The Perl Review*. Papers 2 and 3 prove, using respectively Rice's theorem and direct reduction to the halting problem, that the parsing of Perl programs is in general undecidable.

- [27] Marty Hall, 1995, Lecture Notes: Macros (<http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.html>), PostScript version (<http://www.apl.jhu.edu/~hall/Lisp-Notes/Macros.ps>)
- [28] Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 18-19
- [29] Andrew Cooke. "Introduction To Computer Languages" (<http://www.acooke.org/comp-lang.html>). . Retrieved 13 July 2012.
- [30] Specifically, instantiations of generic types are inferred for certain expression forms. Type inference in Generic Java—the research language that provided the basis for Java 1.5's bounded parametric polymorphism extensions—is discussed in two informal manuscripts from the Types mailing list: Generic Java type inference is unsound (<http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00849.html>) (Alan Jeffrey, 17 December 2001) and Sound Generic Java type inference (<http://www.seas.upenn.edu/~sweirich/types/archive/1999-2003/msg00921.html>) (Martin Odersky, 15 January 2002). C#'s type system is similar to Java's, and uses a similar partial type inference scheme.
- [31] "Revised Report on the Algorithmic Language Scheme" (<http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-4.html>). 20 February 1998. . Retrieved 9 June 2006.
- [32] Luca Cardelli and Peter Wegner. "On Understanding Types, Data Abstraction, and Polymorphism" (<http://citeseer.ist.psu.edu/cardelli85understanding.html>). *Manuscript (1985)*. . Retrieved 9 June 2006.
- [33] Éric Lévénez (2011). "Computer Languages History" (<http://www.levenez.com/lang/>). .
- [34] Jing Huang. "Artificial Language vs. Natural Language" (http://www.cs.cornell.edu/info/Projects/Nuprl/cs611/fall94notes/cn2/subsection3_1-3.html). .
- [35] IBM in first publishing PL/I, for example, rather ambitiously titled its manual *The universal programming language PL/I* (IBM Library; 1966). The title reflected IBM's goals for unlimited subsetting capability: *PL/I is designed in such a way that one can isolate subsets from it satisfying the requirements of particular applications.* ("PL/I" (<http://www.encyclopediaofmath.org/index.php?title=PL/I&oldid=19175>). *Encyclopedia of Mathematics*. . Retrieved 29 June 2006.). Ada and UNCOL had similar early goals.
- [36] Frederick P. Brooks, Jr.: *The Mythical Man-Month*, Addison-Wesley, 1982, pp. 93-94
- [37] Dijkstra, Edsger W. On the foolishness of "natural language programming." (<http://www.cs.utexas.edu/users/EWD/transcriptions/EWD06xx/EWD667.html>) EWD667.
- [38] Perlis, Alan (September 1982). "Epigrams on Programming" (<http://www-pu.informatik.uni-tuebingen.de/users/klaeren/epigrams.html>). *SIGPLAN Notices Vol. 17, No. 9*. pp. 7-13. .
- [39] Milner, R.; M. Tofte, R. Harper and D. MacQueen. (1997). *The Definition of Standard ML (Revised)*. MIT Press. ISBN 0-262-63181-4.
- [40] Kelsey, Richard; William Clinger and Jonathan Rees (February 1998). "Section 7.2 Formal semantics" (http://www.schemers.org/Documents/Standards/R5RS/HTML/r5rs-Z-H-10.html#%_sec_7.2). *Revised⁵ Report on the Algorithmic Language Scheme*. . Retrieved 9 June 2006.
- [41] ANSI — Programming Language Rexx, X3-274.1996
- [42] "HOPL: an interactive Roster of Programming Languages" (<http://hopl.murdoch.edu.au/>). Australia: Murdoch University. . Retrieved 1 June 2009. "This site lists 8512 languages."
- [43] Abelson, Sussman, and Sussman. "Structure and Interpretation of Computer Programs" (<http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-10.html>). . Retrieved 3 March 2009.
- [44] <http://www.computerweekly.com/Articles/2007/09/11/226631/sslcomputer-weekly-it-salary-survey-finance-boom-drives-it-job.htm>
- [45] "Counting programming languages by book sales" (http://radar.oreilly.com/archives/2006/08/programming_language_trends_1.html). Radar.oreilly.com. 2 August 2006. . Retrieved 3 December 2010.
- [46] Bieman, J.M.; Murdock, V., Finding code on the World Wide Web: a preliminary investigation, Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation, 2001
- [47] "Programming Language Popularity" (<http://www.langpop.com/>). Langpop.com. . Retrieved 3 December 2010.
- [48] Carl A. Gunter, *Semantics of Programming Languages: Structures and Techniques*, MIT Press, 1992, ISBN 0-262-57095-5, p. 1
- [49] "TUNES: Programming Languages" (http://tunes.org/wiki/programming_20languages.html). .
- [50] Wirth, Niklaus (1993). "Recollections about the development of Pascal" (<http://portal.acm.org/citation.cfm?id=155378>). *Proc. 2nd ACM SIGPLAN conference on history of programming languages*: 333–342. doi:10.1145/154766.155378. ISBN 0-89791-570-4. . Retrieved 30 June 2006.
- [51] Benjamin C. Pierce writes:
- "... the lambda calculus has seen widespread use in the specification of programming language features, in language design and implementation, and in the study of type systems."
- Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press. p. 52. ISBN 0-262-16209-1.
- [52] Rojas, Raúl, et al. (2000). "Plankalkül: The First High-Level Programming Language and its Implementation". Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000. (full text) (<http://www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Plankalkuel-Report/Plankalkuel-Report.htm>)
- [53] Linda Null, Julia Lobur, *The essentials of computer organization and architecture*, Edition 2, Jones & Bartlett Publishers, 2006, ISBN 0-7637-3769-0, p. 435
- [54] O'Reilly Media. "History of programming languages" (http://www.oreilly.com/news/graphics/prog_lang_poster.pdf) (PDF). . Retrieved 5 October 2006.

- [55] Frank da Cruz. IBM Punch Cards (<http://www.columbia.edu/acis/history/cards.html>) Columbia University Computing History (<http://www.columbia.edu/acis/history/index.html>).
- [56] Richard L. Wexelblat: *History of Programming Languages*, Academic Press, 1981, chapter XIV.
- [57] François Labelle. "Programming Language Usage Graph" (<http://www.cs.berkeley.edu/~flab/languages.html>). *SourceForge*. . Retrieved 21 June 2006.. This comparison analyzes trends in number of projects hosted by a popular community programming repository. During most years of the comparison, C leads by a considerable margin; in 2006, Java overtakes C, but the combination of C/C++ still leads considerably.
- [58] Hayes, Brian (2006). "The Semicolon Wars". *American Scientist* **94** (4): 299–303.
- [59] Dijkstra, Edsger W. (March 1968). "Go To Statement Considered Harmful" (<http://www.acm.org/classics/oct95/>). *Communications of the ACM* **11** (3): 147–148. doi:10.1145/362929.362947. . Retrieved 29 June 2006.
- [60] Tetsuro Fujise, Takashi Chikayama Kazuaki Rokusawa, Akihiko Nakase (December 1994). "KLIC: A Portable Implementation of KL1" *Proc. of FGCS '94, ICOT Tokyo*, December 1994. KLIC is a portable implementation of a concurrent logic programming language [[KL1 (<http://www.icot.or.jp/ARCHIVE/HomePage-E.html>)].]
- [61] Jim Bender (15 March 2004). "Mini-Bibliography on Modules for Functional Programming Languages" (<http://readscheme.org/modules/>). *ReadScheme.org*. . Retrieved 27 September 2006.
- [62] Wall, *Programming Perl* ISBN 0-596-00027-8 p.66

Further reading

- Abelson, Harold; Sussman, Gerald Jay (1996). *[[Structure and Interpretation of Computer Programs* (<http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-4.html>)] (2nd ed.). MIT Press.
- Raphael Finkel: *Advanced Programming Language Design* (<http://www.nondot.org/sabre/Mirrored/AdvProgLangDesign/>), Addison Wesley 1995.
- Daniel P. Friedman, Mitchell Wand, Christopher T. Haynes: *Essentials of Programming Languages*, The MIT Press 2001.
- Maurizio Gabbrielli and Simone Martini: "Programming Languages: Principles and Paradigms", Springer, 2010.
- David Gelernter, Suresh Jagannathan: *Programming Linguistics*, The MIT Press 1990.
- Ellis Horowitz (ed.): *Programming Languages, a Grand Tour* (3rd ed.), 1987.
- Ellis Horowitz: *Fundamentals of Programming Languages*, 1989.
- Shriram Krishnamurthi: *Programming Languages: Application and Interpretation*, online publication (<http://www.cs.brown.edu/~sk/Publications/Books/ProgLangs/>).
- Bruce J. MacLennan: *Principles of Programming Languages: Design, Evaluation, and Implementation*, Oxford University Press 1999.
- John C. Mitchell: *Concepts in Programming Languages*, Cambridge University Press 2002.
- Benjamin C. Pierce: *Types and Programming Languages*, The MIT Press 2002.
- Terrence W. Pratt and Marvin V. Zelkowitz: *Programming Languages: Design and Implementation* (4th ed.), Prentice Hall 2000.
- Peter H. Salus. *Handbook of Programming Languages* (4 vols.). Macmillan 1998.
- Ravi Sethi: *Programming Languages: Concepts and Constructs*, 2nd ed., Addison-Wesley 1996.
- Michael L. Scott: *Programming Language Pragmatics*, Morgan Kaufmann Publishers 2005.
- Robert W. Sebesta: *Concepts of Programming Languages*, 9th ed., Addison Wesley 2009.
- Franklyn Turbak and David Gifford with Mark Sheldon: *Design Concepts in Programming Languages*, The MIT Press 2009.
- Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*, The MIT Press 2004.
- David A. Watt. *Programming Language Concepts and Paradigms*. Prentice Hall 1990.
- David A. Watt and Muffy Thomas. *Programming Language Syntax and Semantics*. Prentice Hall 1991.
- David A. Watt. *Programming Language Processors*. Prentice Hall 1993.
- David A. Watt. *Programming Language Design Concepts*. John Wiley & Sons 2004.

External links

- 99 Bottles of Beer (<http://www.99-bottles-of-beer.net/>) A collection of implementations in many languages.
- Computer Programming Languages (<http://www.dmoz.org/Computers/Programming/Languages/>) at the Open Directory Project

Programming paradigm

A **programming paradigm** is a fundamental style of computer programming. There are four main paradigms : object-oriented, imperative, functional and logic programming.^[1] Their foundations are distinct models of computation : Turing machine for object-oriented and imperative programming, lambda calculus for functional programming, and first order logic for logic programming.

Overview

A *programming model* is an abstraction of a computer system. For example, the "von Neumann model" is a model used in traditional sequential computers. For parallel computing, there are many possible models typically reflecting different ways processors can be interconnected. The most common are based on shared memory, distributed memory with message passing, or a hybrid of the two.

A programming language can support multiple paradigms. For example, programs written in C++ or Object Pascal can be purely procedural, or purely object-oriented, or contain elements of both paradigms. Software designers and programmers decide how to use those paradigm elements.

In object-oriented programming, programmers can think of a program as a collection of interacting objects, while in functional programming a program can be thought of as a sequence of stateless function evaluations. When programming computers or systems with many processors, process-oriented programming allows programmers to think about applications as sets of concurrent processes acting upon logically shared data structures.

Just as different groups in software engineering advocate different *methodologies*, different programming languages advocate different *programming paradigms*. Some languages are designed to support one particular paradigm (Smalltalk supports object-oriented programming, Haskell supports functional programming), while other programming languages support multiple paradigms (such as Object Pascal, C++, Java, C#, Visual Basic, Common Lisp, Scheme, Perl, Python, Ruby, Oz and F#).

Many programming paradigms are as well known for what techniques they *forbid* as for what they enable. For instance, pure functional programming disallows the use of side-effects, while structured programming disallows the use of the goto statement. Partly for this reason, new paradigms are often regarded as doctrinaire or overly rigid by those accustomed to earlier styles.^[2] Avoiding certain techniques can make it easier to prove theorems about a program's correctness—or simply to understand its behavior.

Multi-paradigm programming language

A *multi-paradigm programming language* is a programming language that supports more than one programming paradigm. As Leda designer Timothy Budd puts it: "The idea of a multiparadigm language is to provide a framework in which programmers can work in a variety of styles, freely intermixing constructs from different paradigms." The design goal of such languages is to allow programmers to use the best tool for a job, admitting that no one paradigm solves all problems in the easiest or most efficient way.

One example is C#, which includes imperative and object-oriented paradigms as well as some support for functional programming through type inference, anonymous functions and Language Integrated Query. Some other ones are F# and Scala, which provides similar functionality to C# but also includes full support for functional programming

(including currying, pattern matching, algebraic data types, lazy evaluation, tail recursion, immutability, etc.). Perhaps the most extreme example is Oz, which has subsets that are logic (Oz descends from logic programming), a functional, an object-oriented, a dataflow concurrent, and other language paradigms. Oz was designed over a ten-year period to combine in a harmonious way concepts that are traditionally associated with different programming paradigms. Lisp, while often taught as a functional language, is known for its malleability and thus its ability to engulf many paradigms. A programming paradigm provides for the programmer the means and structure for the execution of a program.

History

The lowest level programming paradigms are machine code, which directly represents the instructions (the contents of program memory) as a sequence of numbers, and assembly language where the machine instructions are represented by mnemonics and memory addresses can be given symbolic labels. These are sometimes called first- and second-generation languages. In the 1960s assembler languages were developed to support library COPY and quite sophisticated conditional macro generation and pre-processing capabilities, CALL to (subroutines), external variables and common sections (globals), enabling significant code re-use and isolation from hardware specifics via use of logical operators such as READ/WRITE/GET/PUT. Assembly was, and still is, used for time critical systems and frequently in embedded systems as it gives the most direct control of what the machine actually does.

The next advance was the development of *procedural languages*. These third-generation languages (the first described as high-level languages) use vocabulary related to the problem being solved. For example,

- C - developed c. 1970 at Bell Labs
- COBOL (Common Business Oriented Language) - uses terms like file, move and copy.
- FORTRAN (FORmula TRANslation) - using mathematical language terminology, it was developed mainly for scientific and engineering problems.
- ALGOL (ALGOrithmic Language) - focused on being an appropriate language to define algorithms, while using mathematical language terminology and targeting scientific and engineering problems just like FORTRAN.
- PL/I (Programming Language One) - a hybrid commercial/scientific general purpose language supporting pointers.
- BASIC (Beginners All purpose Symbolic Instruction Code) - was developed to enable more people to write programs.

All these languages follow the procedural paradigm. That is, they describe, step by step, exactly the procedure that should, according to the particular programmer at least, be followed to solve a specific problem. The efficacy and efficiency of any such solution are both therefore entirely subjective and highly dependent on that programmer's experience, inventiveness and ability.

Later, *object-oriented languages* (like Simula, Smalltalk, C++, Eiffel and Java) were created. In these languages, data, and methods of manipulating the data, are kept as a single unit called an object. The only way that a user can access the data is via the object's 'methods' (subroutines). Because of this, the internal workings of an object may be changed without affecting any code that uses the object. There is still some controversy by notable programmers such as Alexander Stepanov, Richard Stallman^[3] and others, concerning the efficacy of the OOP paradigm versus the procedural paradigm. The necessity of every object to have associative methods leads some skeptics to associate OOP with software bloat. Polymorphism was developed as one attempt to resolve this dilemma.

Since object-oriented programming is considered a paradigm, not a language, it is possible to create even an object-oriented assembler language. High Level Assembly (HLA) is an example of this that fully supports advanced data types and object-oriented assembly language programming - despite its early origins. Thus, differing programming paradigms can be thought of as more like 'motivational memes' of their advocates - rather than necessarily representing progress from one level to the next. Precise comparisons of the efficacy of competing paradigms are frequently made more difficult because of new and differing terminology applied to similar (but not

identical) entities and processes together with numerous implementation distinctions across languages.

Within imperative programming, which is based on procedural languages, an alternative to the computer-centered hierarchy of structured programming is literate programming, which structures programs instead as a human-centered web, as in a hypertext essay – documentation is integral to the program, and the program is structured following the logic of prose exposition, rather than compiler convenience.

Independent of the imperative branch, *declarative programming* paradigms were developed. In these languages the computer is told what the problem is, not how to solve the problem - the program is structured as a collection of properties to find in the expected result, not as a procedure to follow. Given a database or a set of rules, the computer tries to find a solution matching all the desired properties. The archetypical example of a declarative language is the fourth generation language SQL, as well as the family of functional languages and logic programming.

Functional programming is a subset of declarative programming. Programs written using this paradigm use functions, blocks of code intended to behave like mathematical functions. Functional languages discourage changes in the value of variables through assignment, making a great deal of use of recursion instead.

The *logic programming* paradigm views computation as automated reasoning over a corpus of knowledge. Facts about the problem domain are expressed as logic formulae, and programs are executed by applying inference rules over them until an answer to the problem is found, or the collection of formulae is proved inconsistent.

Programming paradigms

This is a list of programming paradigms organized by relationship:

- Action
- Agent-oriented
- Aspect-oriented
- Automata-based
- Component-based
 - Flow-based
 - Pipelined
- Concatenative
- Concurrent computing
 - Relativistic programming
- Data-driven
- Declarative (contrast: Imperative)
 - Constraint
 - Dataflow
 - Cell-oriented (spreadsheets)
 - Reactive
 - Intensional
 - Functional
 - Logic
 - Abductive logic
 - Answer set
 - Constraint logic
 - Functional logic
 - Inductive logic
- End-user programming
- Event-driven

- Service-oriented
- Time-driven
- Expression-oriented
- Feature-oriented
- Function-level (contrast: Value-level)
- Generic
- Imperative (contrast: Declarative)
 - Procedural
- Language-oriented
 - Discipline-specific
 - Domain-specific
 - Grammar-oriented
 - Dialecting
 - Intentional
- Metaprogramming
 - Automatic
 - Reflective
 - Attribute-oriented
 - Template
 - Policy-based
- Non-structured (contrast: Structured)
 - Array
- Nondeterministic
- Parallel computing
 - Process-oriented
- Programming in the large and small
- Semantic
- Structured (contrast: Non-structured)
 - Modular (contrast: Monolithic)
 - Object-oriented
 - By separation of concerns:
 - Aspect-oriented
 - Role-oriented
 - Subject-oriented
 - Class-based
 - Prototype-based
 - Recursive
- Value-level (contrast: Function-level)

References

- [1] Nørmark , Kurt. *Overview of the four main programming paradigms* (http://people.cs.aau.dk/~normark/prog3-03/html/notes/paradigms_themes-paradigm-overview-section.html). Aalborg University, 9 May 2011. Retrieved 22 September 2012.
- [2] Frank Rubin published a criticism of Dijkstra's letter in the March 1987 CACM where it appeared under the title '*GOTO Considered Harmful' Considered Harmful*'. Frank Rubin (March 1987). "'GOTO Considered Harmful' Considered Harmful" (<http://www.ecn.purdue.edu/ParaMount/papers/rubin87goto.pdf>) (PDF). *Communications of the ACM* **30** (3): 195–196. doi:10.1145/214748.315722. .
- [3] "Mode inheritance, cloning, hooks & OOP (Google Groups Discussion)" (http://groups.google.com/group/comp.emacs.xemacs/browse_thread/thread/d0af257a2837640c/37f251537fafbb03?lnk=st&q=%22Richard+Stallman%22+oop&rnum=5&hl=en#37f251537fafbb03).

External links

- Classification of the principal programming paradigms (<http://www.info.ucl.ac.be/~pvr/paradigms.html>)

Prehistoric

Procedural programming

Procedural programming can sometimes be used as a synonym for imperative programming (specifying the steps the program must take to reach the desired state), but can also refer (as in this article) to a programming paradigm, derived from structured programming, based upon the concept of the *procedure call*. Procedures, also known as routines, subroutines, methods, or functions (not to be confused with mathematical functions, but similar to those used in functional programming), simply contain a series of computational steps to be carried out. Any given procedure might be called at any point during a program's execution, including by other procedures or itself.^[1]

Procedures and modularity

Modularity is generally desirable, especially in large, complicated programs. Inputs are usually specified syntactically in the form of *arguments* and the outputs delivered as *return values*.

Scoping is another technique that helps keep procedures strongly modular. It prevents the procedure from accessing the variables of other procedures (and vice-versa), including previous instances of itself, without explicit authorization.

Less modular procedures, often used in small or quickly written programs, tend to interact with a large number of variables in the execution environment, which other procedures might also modify.

Because of the ability to specify a simple interface, to be self-contained, and to be reused, procedures are a convenient vehicle for making pieces of code written by different people or different groups, including through programming libraries.

Comparison with imperative programming

Procedural programming languages are also imperative languages, because they make explicit references to the state of the execution environment. This could be anything from *variables* (which may correspond to processor registers) to something like the position of the "turtle" in the Logo programming language.

Comparison with object-oriented programming

The focus of procedural programming is to break down a programming task into a collection of variables, data structures, and subroutines, whereas in object-oriented programming it is to break down a programming task into objects that expose behavior (methods) and data (members or attributes) using interfaces. The most important distinction is whereas procedural programming uses procedures to operate on data structures, object-oriented programming bundles the two together so an "object", which is an instance of a class, operates on its "own" data structure.

Nomenclature varies between the two, although they have similar semantics:

Procedural	Object-oriented
procedure	method
record	object
module	class
procedure call	message

See Algorithms + Data Structures = Programs.

Comparison with functional programming

The principles of modularity and code reuse in practical functional languages are fundamentally the same as in procedural languages, since they both stem from structured programming. So for example:

- Procedures correspond to functions. Both allow the reuse of the same code in various parts of the programs, and at various points of its execution.
- By the same token, procedure calls correspond to function application.
- Functions and their invocations are modularly separated from each other in the same manner, by the use of function arguments, return values and variable scopes.

The main difference between the styles is that functional programming languages remove or at least deemphasize the imperative elements of procedural programming. The feature set of functional languages is therefore designed to support writing programs as much as possible in terms of pure functions:

- Whereas procedural languages model execution of the program as a sequence of imperative commands that may implicitly alter shared state, functional programming languages model execution as the evaluation of complex expressions that only depend on each other in terms of arguments and return values. For this reason, functional programs can have a freer order of code execution, and the languages may offer little control over the order in which various parts of the program are executed. (For example, the arguments to a procedure invocation in Scheme are executed in an arbitrary order.)
- Functional programming languages support (and heavily use) first-class functions, anonymous functions and closures.
- Functional programming languages tend to rely on tail call optimization and higher-order functions instead of imperative looping constructs.

Many functional languages, however, are in fact impurely functional and offer imperative/procedural constructs that allow the programmer to write programs in procedural style, or in a combination of both styles. It is common for input/output code in functional languages to be written in a procedural style.

There do exist a few esoteric functional languages (like Unlambda) that eschew structured programming precepts for the sake of being difficult to program in (and therefore challenging). These languages are the exception to the common ground between procedural and functional languages.

Comparison with logic programming

In logic programming, a program is a set of premises, and computation is performed by attempting to prove candidate theorems. From this point of view, logic programs are declarative, focusing on what the problem is, rather than on how to solve it.

However, the backward reasoning technique, implemented by SLD resolution, used to solve problems in logic programming languages such as Prolog, treats programs as goal-reduction procedures. Thus clauses of the form:

$H :- B_1, \dots, B_n.$

have a dual interpretation, both as procedures

to show/solve H , show/solve B_1 and ... and B_n

and as logical implications:

$B_1 \text{ and } \dots \text{ and } B_n \text{ implies } H.$

Experienced logic programmers use the procedural interpretation to write programs that are effective and efficient, and they use the declarative interpretation to help ensure that programs are correct.

References

- [1] "Welcome to IEEE Xplore 2.0: Use of procedural programming languages for controlling production systems" (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=120848). ieeexplore.ieee.org.. Retrieved 2008-04-06.

External links

- Procedural Languages (<http://www.dmoz.org/Computers/Programming/Languages/Procedural/>) at the Open Directory Project

Structured programming

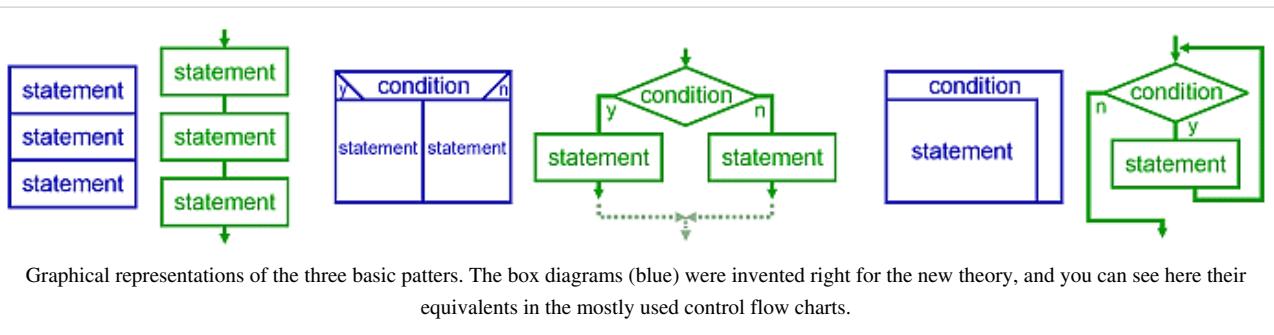
Structured programming is a programming paradigm aimed on improving the clarity, quality, and development time of a computer program by making extensive use of subroutines, block structures and for and while loops – in contrast to using simple tests and jumps such as the *goto* statement which could lead to "spaghetti code" which is both difficult to follow and to maintain.

It emerged in the 1960s, particularly from work by Böhm and Jacopini,^[1] and a famous letter, *Go To Statement Considered Harmful*, from Edsger Dijkstra in 1968^[2]—and was bolstered theoretically by the structured program theorem, and practically by the emergence of languages such as ALGOL with suitably rich control structures.

Low-level structure programming

At a low level, structured programs are often composed of simple, hierarchical program flow structures. These are sequence, selection, and repetition:

- "Sequence" refers to an ordered execution of statements.
- In "selection" one of a number of statements is executed depending on the state of the program. This is usually expressed with keywords such as *if..then..else..endif*, *switch*, or *case*. In some languages keywords cannot be written verbatim, but must be stropped.
- In "repetition" a statement is executed until the program reaches a certain state, or operations have been applied to every element of a collection. This is usually expressed with keywords such as *while*, *repeat*, *for* or *do..until*. Often it is recommended that each loop should only have one entry point (and in the original structural programming, also only one exit point, and a few languages enforce this).



A language is described as **block-structured** when it has a syntax for enclosing structures between bracketed keywords, such as an if-statement bracketed by *if..fi* as in ALGOL 68, or a code section bracketed by *BEGIN..END*, as in PL/I - or the curly braces *{ . . . }* of C and many later languages.

Structured programming languages

It is possible to do structured programming in any programming language, though it is preferable to use something like a procedural programming language. Some of the languages initially used for structured programming languages include: ALGOL, Pascal, PL/I and Ada – but most new procedural programming languages since that time have included features to encourage structured programming, and sometimes deliberately left out features^[3] in an effort to make unstructured programming more difficult.

History

Theoretical foundation

The structured program theorem provides the theoretical basis of structured programming. It states that three ways of combining programs—sequencing, selection, and iteration—are sufficient to express any computable function. This observation did not originate with the structured programming movement; these structures are sufficient to describe the instruction cycle of a central processing unit, as well as the operation of a Turing machine. Therefore a processor is always executing a "structured program" in this sense, even if the instructions it reads from memory are not part of a structured program. However, authors usually credit the result to a 1966 paper by Böhm and Jacopini, possibly because Dijkstra cited this paper himself. The structured program theorem does not address how to write and analyze a usefully structured program. These issues were addressed during the late 1960s and early 1970s, with major contributions by Dijkstra, Robert W. Floyd, Tony Hoare, and David Gries.

Debate

P. J. Plauger, an early adopter of structured programming, described his reaction to the structured program theorem:

Us converts waved this interesting bit of news under the noses of the unreconstructed assembly-language programmers who kept trotting forth twisty bits of logic and saying, 'I betcha can't structure this.' Neither the proof by Böhm and Jacopini nor our repeated successes at writing structured code brought them around one day sooner than they were ready to convince themselves.^[4]

Donald Knuth accepted the principle that programs must be written with provability in mind, but he disagreed (and still disagrees) with abolishing the GOTO statement. In his 1974 paper, "Structured Programming with Goto Statements", he gave examples where he believed that a direct jump leads to clearer and more efficient code without sacrificing provability. Knuth proposed a looser structural constraint: It should be possible to draw a program's flow chart with all forward branches on the left, all backward branches on the right, and no branches crossing each other. Many of those knowledgeable in compilers and graph theory have advocated allowing only reducible flow graphs.

Structured programming theorists gained a major ally in the 1970s after IBM researcher Harlan Mills applied his interpretation of structured programming theory to the development of an indexing system for the *New York Times* research file. The project was a great engineering success, and managers at other companies cited it in support of adopting structured programming, although Dijkstra criticized the ways that Mills's interpretation differed from the published work.

As late as 1987 it was still possible to raise the question of structured programming in a computer science journal. Frank Rubin did so in that year with a letter, "'GOTO considered harmful' considered harmful." Numerous objections followed, including a response from Dijkstra that sharply criticized both Rubin and the concessions other writers made when responding to him.

Outcome

By the end of the 20th century nearly all computer scientists were convinced that it is useful to learn and apply the concepts of structured programming. High-level programming languages that originally lacked programming structures, such as FORTRAN, COBOL, and BASIC, now have them.

Common deviations

Exception handling

Although there is almost never a reason to have multiple points of entry to a subprogram, multiple exits are often used to reflect that a subprogram may have no more work to do, or may have encountered circumstances that prevent it from continuing.

A typical example of a simple procedure would be reading data from a file and processing it:

```
open file;
while (reading not finished) {
    read some data;
    if (error) {
        stop the subprogram and inform rest of the program about the error;
    }
    process read data;
}
finish the subprogram;
```

The "stop and inform" may be achieved by throwing an exception, second return from the procedure, labelled loop break, or even a goto. Many languages have instructions that allow unstructured exiting from loops and procedures. C allows multiple paths to a structure's exit (such as "continue", "break", and "return"), newer languages have also "labelled breaks" (similar to the former, but allowing breaking out of more than just the innermost loop) and exceptions.

The procedure above breaks the rules of Dijkstra's structured programming because the while loop can terminate in a way that is not specified in the while condition (the "reading not finished"), and because it has two exit points (the end of the procedure and the "stop" on error). Re-coding this to use a structured while loop and a single point of exit is trivial:

```
open file;
while (no error) and (reading not finished) {
    read some data;
    if (no error) {
        process read data;
    }
}
if (error) {
    inform rest of the program about the error;
}
else {
    finish the subprogram;
}
```

This removes the unstructured termination of the while loop and gives the procedure a single exit point without any change to the overall function.

State machines

Some programs, particularly parsers and communications protocols, have a number of states that follow each other in a way that is not easily reduced to the basic structures. It is possible to structure these systems by making each state-change a separate subprogram and using a variable to indicate the active state (see trampoline). However, some programmers (including Knuth) prefer to implement the state-changes with a jump to the new state.

References

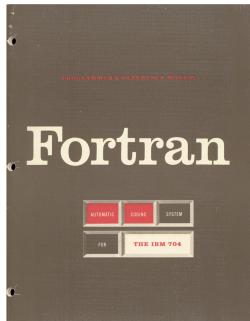
1. Edsger Dijkstra, *Notes on Structured Programming* [5], pg. 6
 2. Böhm, C. and Jacopini, G.: *Flow diagrams, Turing machines and languages with only two formation rules*, CACM 9(5), 1966.
 3. Michael A. Jackson, *Principles of Program Design*, Academic Press, London, 1975.
 4. O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare *Structured Programming*, Academic Press, London, 1972 ISBN 0-12-200550-3
 - this volume includes an expanded version of the *Notes on Structured Programming*, above, including an extended example of using the structured approach to develop a backtracking algorithm to solve the 8 Queens problem.
 - a pdf version is in the ACM Classic Books Series [6]
 - Note that the third chapter of this book, by Dahl, describes an approach which is easily recognized as Object Oriented Programming. It can be seen as another way to "usefully structure" a program to aid in showing that it is correct.
- [1] Böhm, Jacopini. "Flow diagrams, turing machines and languages with only two formation rules" Comm. ACM, 9(5):366-371, May 1966
- [2] Edsger Dijkstra (March 1968). "Go To Statement Considered Harmful" (PDF). *Communications of the ACM* 11 (3): 147–148.
doi:10.1145/362929.362947. "The unbridled use of the go to statement has as an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. ... The go to statement as it stands is just too primitive, it is too much an invitation to make a mess of one's program."
- [3] GOTO for example
- [4] Plauger, P. J. (February 12, 1993). *Programming on Purpose, Essays on Software Design* (1 ed.). Prentice-Hall. p. 25.
ISBN 978-0-13-721374-0.
- [5] <http://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF>
- [6] <http://portal.acm.org/citation.cfm?id=1243380&jmp=cit&coll=portal&dl=GUIDE&CFID=://www.acm.org/publications/&CFTOKEN=www.acm.org/publications/#CIT>

External links

- BPStruct (<http://code.google.com/p/bpstruct/>) - A tool to structure concurrent systems (programs, process models)
- Algobuild: Free editor for structured flow charts and pseudo code (EN - IT) (<http://algobuild.com/>)
- J. Darlington, M. Ghanem, H. W. To (1993), "Structured Parallel Programming" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.37.4610>), *In Programming Models for Massively Parallel Computers. IEEE Computer Society Press. 1993*

Fortran

Fortran



The Fortran Automatic Coding System for the IBM 704 (15 October 1956), the first Programmer's Reference Manual for Fortran

Paradigm(s)	multi-paradigm: structured, imperative (procedural, object-oriented), generic
Appeared in	1957
Designed by	John Backus
Developer	John Backus & IBM
Stable release	Fortran 2008 (ISO/IEC 1539-1:2010) (2010)
Typing discipline	strong, static, manifest
Major implementations	Absoft, Cray, GFortran, G95, IBM, Intel, Lahey/Fujitsu, Open Watcom, PathScale, PGI, Silverfrost, Oracle, XL Fortran, Visual Fortran, others
Influenced by	Speedcoding
Influenced	ALGOL 58, BASIC, C, PL/I, PACT I, MUMPS, Ratfor
Usual filename extensions	.f, .for, .f90, .f95

Fortran (previously **FORTRAN**) is a general-purpose, imperative programming language that is especially suited to numeric computation and scientific computing. Originally developed by IBM at their campus in south San Jose, California^[1] in the 1950s for scientific and engineering applications, Fortran came to dominate this area of programming early on and has been in continual use for over half a century in computationally intensive areas such as numerical weather prediction, finite element analysis, computational fluid dynamics, computational physics and computational chemistry. It is one of the most popular languages in the area of high-performance computing^[2] and is the language used for programs that benchmark and rank the world's fastest supercomputers.

Fortran (a blend derived from *The IBM Mathematical Formula Translating System*) encompasses a lineage of versions, each of which evolved to add extensions to the language while usually retaining compatibility with previous versions. Successive versions have added support for structured programming and processing of character-based data (FORTRAN 77), array programming, modular programming and generic programming (Fortran 90), high performance Fortran (Fortran 95), object-oriented programming (Fortran 2003) and concurrent programming (Fortran 2008).

Capitalization

The names of earlier versions of the language through FORTRAN 77 were conventionally spelled in all-caps (FORTRAN 77 was the version in which the use of lowercase letters in keywords was strictly nonstandard). The capitalization has been dropped in referring to newer versions beginning with Fortran 90. The official language standards now refer to the language as "Fortran". Because the capitalization was never completely consistent in actual usage, this article adopts the convention of using the all-caps *FORTRAN* in referring to versions of FORTRAN through FORTRAN 77 and the title-caps *Fortran* in referring to versions of Fortran from Fortran 90 onward. This convention is reflected in the capitalization of *FORTRAN* in the ANSI X3.9-1966 (FORTRAN 66) and ANSI X3.9-1978 (FORTRAN 77) standards and the title caps *Fortran* in the ANSI X3.198-1992 (Fortran 90), ISO/IEC 1539-1:1997 (Fortran 95) and ISO/IEC 1539-1:2004 (Fortran 2003) standards.

History

In late 1953, John W. Backus submitted a proposal to his superiors at IBM to develop a more practical alternative to assembly language for programming their IBM 704 mainframe computer. Backus' historic FORTRAN team consisted of programmers Richard Goldberg, Sheldon F. Best, Harlan Herrick, Peter Sheridan, Roy Nutt, Robert Nelson, Irving Ziller, Lois Haibt, and David Sayre.^[3] Its concepts included easier entry of equations into a computer, an idea developed by J. Halcombe Laning and demonstrated in his GEORGE compiler of 1952.^[4]



An IBM 704 mainframe

A draft specification for *The IBM Mathematical Formula Translating System* was completed by mid-1954. The first manual for FORTRAN appeared in October 1956, with the first FORTRAN compiler delivered in April 1957. This was the first optimizing compiler, because customers were reluctant to use a high-level programming language unless its compiler could generate code whose performance was comparable to that of hand-coded assembly language.^[5]

While the community was skeptical that this new method could possibly outperform hand-coding, it reduced the number of programming statements necessary to operate a machine by a factor of 20, and quickly gained acceptance. John Backus said during a 1979 interview with *Think*, the IBM employee magazine, "Much of my work has come from being lazy. I didn't like writing programs, and so, when I was working on the IBM 701, writing programs for computing missile trajectories, I started work on a programming system to make it easier to write programs."^[6]

The language was widely adopted by scientists for writing numerically intensive programs, which encouraged compiler writers to produce compilers that could generate faster and more efficient code. The inclusion of a complex number data type in the language made Fortran especially suited to technical applications such as electrical engineering.

By 1960, versions of FORTRAN were available for the IBM 709, 650, 1620, and 7090 computers. Significantly, the increasing popularity of FORTRAN spurred competing computer manufacturers to provide FORTRAN compilers for their machines, so that by 1963 over 40 FORTRAN compilers existed. For these reasons, FORTRAN is considered to be the first widely used programming language supported across a variety of computer architectures.

The development of FORTRAN paralleled the early evolution of compiler technology, and many advances in the theory and design of compilers were specifically motivated by the need to generate efficient code for FORTRAN programs.

FORTRAN

The initial release of FORTRAN for the IBM 704 contained 32 statements, including:

- DIMENSION and EQUIVALENCE statements
- Assignment statements
- Three-way *arithmetic IF* statement.
- IF statements for checking exceptions (ACCUMULATOR OVERFLOW, QUOTIENT OVERFLOW, and DIVIDE CHECK); and IF statements for manipulating sense switches and sense lights
- GOTO, computed GOTO, ASSIGN, and assigned GOTO
- DO loops
- Formatted I/O: FORMAT, READ, READ INPUT TAPE, WRITE, WRITE OUTPUT TAPE, PRINT, and PUNCH
- Unformatted I/O: READ TAPE, READ DRUM, WRITE TAPE, and WRITE DRUM
- Other I/O: END FILE, REWIND, and BACKSPACE
- PAUSE, STOP, and CONTINUE
- FREQUENCY statement (for providing optimization hints to the compiler).

The arithmetic IF statement was similar to a three-way branch instruction on the IBM 704. However, the 704 branch instructions all contained only one destination address (e.g., TZE — Transfer AC Zero, TNZ — Transfer AC Not Zero, TPL — Transfer AC Plus, TMI — Transfer AC Minus). The machine (and its successors in the 700/7000 series) did have a three-way *skip* instruction (CAS — Compare AC with Storage), but using this instruction to implement the IF would consume 4 instruction words, require the constant Zero in a word of storage, and take 3 machine cycles to execute; using the Transfer instructions to implement the IF could be done in 1 to 3 instruction words, required no constants in storage, and take 1 to 3 machine cycles to execute. An optimizing compiler like FORTRAN would most likely select the more compact and usually faster Transfers instead of the Compare (use of Transfers also allowed the FREQUENCY statement to optimize IFs, which could not be done using the Compare). Also the Compare considered -0 and +0 to be different values while the Transfer Zero and Transfer Not Zero considered them to be the same.

The FREQUENCY statement in FORTRAN was used originally and optionally to give branch probabilities for the three branch cases of the Arithmetic IF statement to bias the way code was generated and order of the basic blocks of code generated, in the global optimisation sense, were arranged in memory for optimality. The first FORTRAN compiler used this weighting to do a Monte Carlo simulation of the run-time generated code at compile time. It was very sophisticated for its time. This technique is documented in the original article in 1957 on the first FORTRAN compiler implementation by J. Backus et al. Many years later, the FREQUENCY statement had no effect on the code, and was treated as a comment statement, since the compilers no longer did this kind of compile-time simulation.

Below is a part of the 1957 paper, "The FORTRAN Automatic Coding System" by Backus et al., with this snippet on the FREQUENCY statement and its use in a compile-time Monte Carlo simulation of the run-time to optimise the code generated. Quoting ...

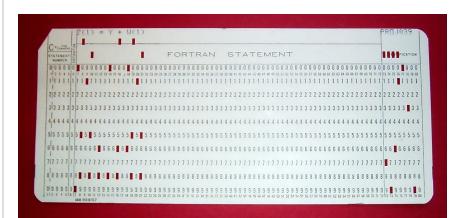
The fundamental unit of program is the basic block; a basic block is a stretch of program which has a single entry point and a single exit point. The purpose of section 4 is to prepare for section 5 a table of predecessors (PRED table) which enumerates the basic blocks and lists for every basic block each of the basic blocks which can be its immediate predecessor in flow, together with the absolute frequency of each such basic block link. This table is obtained by an actual "execution" of the program in Monte-Carlo fashion, in which the outcome of conditional transfers arising out of IF-type statements and computed GO TO'S is determined by a random number generator suitably weighted according to

whatever FREQUENCY statements have been provided.

Fixed layout

Before the development of disk files, text editors and terminals, programs were most often entered on a keypunch keyboard onto 80 column punched cards, one line to a card. The resulting deck of cards would be fed into a card reader to be compiled. *See Computer programming in the punched card era.*

Originally Fortran programs were written in a fixed column format. A letter "C" in column 1 caused the entire card to be treated as a comment and ignored by the compiler. Otherwise, the card was divided into four fields. Columns 1 to 5 were the label field: a sequence of digits here was taken as a label for the purpose of a GOTO or a FORMAT reference in a WRITE or READ statement. Column 6 was a continuation field: a non-blank character here caused the card to be taken as a continuation of the statement on the previous card. Columns 7 to 72 served as the statement field. Columns 73 to 80 were ignored, so they could be used for identification information. One such use was punching a sequence number which could be used to re-order cards if a stack of cards was dropped, though in practice this was reserved for stable, production programs. An IBM 519 could be used to copy a program deck and add sequence numbers. Some early compilers, e.g. the IBM 650's, had additional restrictions.^[7] Later compilers relaxed most fixed format restrictions.



FORTRAN code on a punched card, showing the specialized uses of columns 1-5, 6 and 73-80.

Within the statement field, blanks were generally ignored, allowing the programmer to omit space between tokens for brevity, or include spaces within identifiers for clarity (for example, AVG OF X was a valid identifier, and equivalent to AVGOFX).

Recursion

Early FORTRAN compilers did not support recursion in subroutines. Early computer architectures did not support the concept of a stack, and when they did directly support subroutine calls, the return location was often stored in a single fixed location adjacent to the subroutine code, which does not permit a subroutine to be called again before a previous call of the subroutine has returned. Although not specified in Fortran 77, many F77 compilers supported recursion as an option, while it became a standard in Fortran 90.^[8]

FORTRAN II

IBM's *FORTRAN II* appeared in 1958. The main enhancement was to support procedural programming by allowing user-written subroutines and functions which returned values, with parameters passed by reference. The COMMON statement provided a way for subroutines to access common (or global) variables. Six new statements were introduced:

- SUBROUTINE, FUNCTION, and END
- CALL and RETURN
- COMMON

Over the next few years, FORTRAN II would also add support for the DOUBLE PRECISION and COMPLEX data types.

Simple FORTRAN II program

This program, for Heron's formula, reads one data card containing three 5-digit integers A, B, and C as input. If A, B, and C cannot represent the sides of a triangle in plane geometry, then the program's execution will end with an error code of "STOP 1". Otherwise, an output line will be printed showing the input values for A, B, and C, followed by the computed AREA of the triangle as a floating-point number with 2 digits after the decimal point.

```
C AREA OF A TRIANGLE WITH A STANDARD SQUARE ROOT FUNCTION
C INPUT - CARD READER UNIT 5, INTEGER INPUT
C OUTPUT - LINE PRINTER UNIT 6, REAL OUTPUT
C INPUT ERROR DISPLAY ERROR OUTPUT CODE 1 IN JOB CONTROL LISTING
    READ INPUT TAPE 5, 501, IA, IB, IC
501 FORMAT (3I5)
C IA, IB, AND IC MAY NOT BE NEGATIVE
C FURTHERMORE, THE SUM OF TWO SIDES OF A TRIANGLE
C IS GREATER THAN THE THIRD SIDE, SO WE CHECK FOR THAT, TOO
    IF (IA) 777, 777, 701
701 IF (IB) 777, 777, 702
702 IF (IC) 777, 777, 703
703 IF (IA+IB-IC) 777,777,704
704 IF (IA+IC-IB) 777,777,705
705 IF (IB+IC-IA) 777,777,799
777 STOP 1
C USING HERON'S FORMULA WE CALCULATE THE
C AREA OF THE TRIANGLE
799 S = FLOATF (IA + IB + IC) / 2.0
    AREA = SQRT( S * (S - FLOATF(IA)) * (S - FLOATF(IB)) *
+           (S - FLOATF(IC)))
    WRITE OUTPUT TAPE 6, 601, IA, IB, IC, AREA
601 FORMAT (4H A= ,I5,5H B= ,I5,5H C= ,I5,8H AREA= ,F10.2,
+           13H SQUARE UNITS)
    STOP
    END
```

FORTRAN III

IBM also developed a *FORTRAN III* in 1958 that allowed for inline assembler code among other features; however, this version was never released as a product. Like the 704 FORTRAN and FORTRAN II, FORTRAN III included machine-dependent features that made code written in it unportable from machine to machine. Early versions of FORTRAN provided by other vendors suffered from the same disadvantage.

IBM 1401 FORTRAN

FORTRAN was provided for the IBM 1401 by an innovative 63-pass compiler that ran in only 8k of core. It kept the program in memory and loaded overlays that gradually transformed it, in place, into executable form, as described by Haines et al.^[9] The executable form was not machine language; rather it was interpreted, anticipating UCSD Pascal P-code by two decades.

A FORTRAN coding form, printed on paper and intended to be used by programmers to prepare programs for punching onto cards by keypunch operators.

Now obsolete.

FORTRAN IV

Starting in 1961, as a result of customer demands, IBM began development of a *FORTRAN IV* that removed the machine-dependent features of FORTRAN II (such as `READ INPUT TAPE`), while adding new features such as a `LOGICAL` data type, logical Boolean expressions and the *logical IF statement* as an alternative to the *arithmetic IF statement*. FORTRAN IV was eventually released in 1962, first for the IBM 7030 ("Stretch") computer, followed by versions for the IBM 7090 and IBM 7094.

By 1965, FORTRAN IV was supposed to be compliant with the "standard" being developed by the American Standards Association X3.4.3 FORTRAN Working Group.^[10]

At about this time FORTRAN IV had started to become an important educational tool and implementations such as Waterloo University's WATFOR and WATFIV were created to simplify the complex compile and link processes of earlier compilers.

FORTRAN 66

Perhaps the most significant development in the early history of FORTRAN was the decision by the *American Standards Association* (now ANSI) to form a committee sponsored by BEMA, the Business Equipment Manufacturers Association, to develop an "American Standard Fortran." The resulting two standards, approved in March 1966, defined two languages, *FORTRAN* (based on FORTRAN IV, which had served as a *de facto* standard), and *Basic FORTRAN* (based on FORTRAN II, but stripped of its machine-dependent features). The FORTRAN defined by the first standard, officially denoted X3.9-1966, became known as *FORTRAN 66* (although many continued to refer to it as FORTRAN IV, the language upon which the standard was largely based). FORTRAN 66 effectively became the first "industry-standard" version of FORTRAN. FORTRAN 66 included:

- Main program, SUBROUTINE, FUNCTION, and BLOCK DATA program units
- INTEGER, REAL, DOUBLE PRECISION, COMPLEX, and LOGICAL data types
- COMMON, DIMENSION, and EQUIVALENCE statements
- DATA statement for specifying initial values

- Intrinsic and EXTERNAL (*e.g.*, library) functions
- Assignment statement
- GOTO, assigned GOTO, and computed GOTO statements
- Logical IF and arithmetic (three-way) IF statements
- DO loops
- READ, WRITE, BACKSPACE, REWIND, and ENDFILE statements for sequential I/O
- FORMAT statement
- CALL, RETURN, PAUSE, and STOP statements
- Hollerith constants in DATA and FORMAT statements, and as actual arguments to procedures
- Identifiers of up to six characters in length
- Comment lines

FORTRAN 77

After the release of the FORTRAN 66 standard, compiler vendors introduced a number of extensions to "Standard Fortran", prompting ANSI committee X3J3 in 1969 to begin work on revising the 1966 standard, under sponsorship of CBEMA, the Computer Business Equipment Manufacturers Association (formerly BEMA). Final drafts of this revised standard circulated in 1977, leading to formal approval of the new FORTRAN standard in April 1978. The new standard, known as *FORTRAN 77* and officially denoted X3.9-1978, added a number of significant features to address many of the shortcomings of FORTRAN 66:

- Block IF and END IF statements, with optional ELSE and ELSE IF clauses, to provide improved language support for structured programming
- DO loop extensions, including parameter expressions, negative increments, and zero trip counts
- OPEN, CLOSE, and INQUIRE statements for improved I/O capability
- Direct-access file I/O
- IMPLICIT statement
- CHARACTER data type, with vastly expanded facilities for character input and output and processing of character-based data
- PARAMETER statement for specifying constants
- SAVE statement for persistent local variables
- Generic names for intrinsic functions
- A set of intrinsics (LGE, LGT, LLE, LLT) for lexical comparison of strings, based upon the ASCII collating sequence. (These ASCII functions were demanded by the U.S. Department of Defense, in their conditional approval vote.)

In this revision of the standard, a number of features were removed or altered in a manner that might invalidate previously standard-conforming programs. (*Removal was the only allowable alternative to X3J3 at that time, since the concept of "deprecation" was not yet available for ANSI standards.*) While most of the 24 items in the conflict list (see Appendix A2 of X3.9-1978) addressed loopholes or pathological cases permitted by the previous standard but rarely used, a small number of specific capabilities were deliberately removed, such as:

- Hollerith constants and Hollerith data, such as:

```
GREET = 12HHELLO THERE!
```

- Reading into a H edit (Hollerith field) descriptor in a FORMAT specification.
- Overindexing of array bounds by subscripts.

```
DIMENSION A(10,5)
```

```
Y= A(11,1)
```

- Transfer of control out of and back into the range of a DO loop (also known as "Extended Range").

Variants: Minnesota FORTRAN

Control Data Corporation computers had another version of FORTRAN 77, called Minnesota FORTRAN (MNF), designed especially for student use, with variations in output constructs, special uses of COMMONs and DATA statements, optimizations code levels for compiling, and detailed error listings, extensive warning messages, and debugs.^[11]

Transition to ANSI Standard Fortran

The development of a revised standard to succeed FORTRAN 77 would be repeatedly delayed as the standardization process struggled to keep up with rapid changes in computing and programming practice. In the meantime, as the "Standard FORTRAN" for nearly fifteen years, FORTRAN 77 would become the historically most important dialect. An important practical extension to FORTRAN 77 was the release of MIL-STD-1753 in 1978.^[12] This specification, developed by the U.S. Department of Defense, standardized a number of features implemented by most FORTRAN 77 compilers but not included in the ANSI FORTRAN 77 standard. These features would eventually be incorporated into the Fortran 90 standard.

- DO WHILE and END DO statements
- INCLUDE statement
- IMPLICIT NONE variant of the IMPLICIT statement
- Bit manipulation intrinsic functions, based on similar functions included in Industrial Real-Time Fortran (ANSI/ISA S61.1 (1976))

The IEEE 1003.9 POSIX Standard, released in 1991, provided a simple means for FORTRAN 77 programmers to issue POSIX system calls.^[13] Over 100 calls were defined in the document — allowing access to POSIX-compatible process control, signal handling, file system control, device control, procedure pointing, and stream I/O in a portable manner.

Fortran 90

The much delayed successor to FORTRAN 77, informally known as *Fortran 90* (and prior to that, *Fortran 8X*), was finally released as ISO/IEC standard 1539:1991 in 1991 and an ANSI Standard in 1992. In addition to changing the official spelling from FORTRAN to Fortran, this major revision added many new features to reflect the significant changes in programming practice that had evolved since the 1978 standard:

- Free-form source input, also with lowercase Fortran keywords
- Identifiers up to 31 characters in length
- Inline comments
- Ability to operate on arrays (or array sections) as a whole, thus greatly simplifying math and engineering computations.
 - whole, partial and masked array assignment statements and array expressions, such as
 $X(1:N)=R(1:N)*COS(A(1:N))$
 - WHERE statement for selective array assignment
 - array-valued constants and expressions,
 - user-defined array-valued functions and array constructors.
- RECURSIVE procedures
- Modules, to group related procedures and data together, and make them available to other program units, including the capability to limit the accessibility to only specific parts of the module.
- A vastly improved argument-passing mechanism, allowing interfaces to be checked at compile time
- User-written interfaces for generic procedures
- Operator overloading
- Derived (structured) data types

- New data type declaration syntax, to specify the data type and other attributes of variables
- Dynamic memory allocation by means of the ALLOCATABLE attribute and the ALLOCATE and DEALLOCATE statements
- POINTER attribute, pointer assignment, and NULLIFY statement to facilitate the creation and manipulation of dynamic data structures
- Structured looping constructs, with an END DO statement for loop termination, and EXIT and CYCLE statements for terminating normal DO loop iterations in an orderly way
- SELECT ... CASE construct for multi-way selection
- Portable specification of numerical precision under the user's control
- New and enhanced intrinsic procedures.

Obsolescence and deletions

Unlike the previous revision, Fortran 90 did not delete any features. (*Appendix B.1 says, "The list of deleted features in this standard is empty."*) Any standard-conforming FORTRAN 77 program is also standard-conforming under Fortran 90, and either standard should be usable to define its behavior.

A small set of features were identified as "obsolescent" and expected to be removed in a future standard.

Obsolete feature	Example	Status / Fate in Fortran 95
Arithmetic IF-statement	IF (X) 10, 20, 30	
Non-integer DO parameters or control variables	DO 9 X= 1.7, 1.6, -0.1	Deleted
Shared DO-loop termination or termination with a statement other than END DO or CONTINUE	DO 9 J= 1, 10 DO 9 K= 1, 10 9 L= J + K	
Branching to END IF from outside a block	66 GO TO 77 ; . . . IF (E) THEN ; . . . 77 END IF	Deleted
Alternate return	CALL SUBR(X, Y *100, *200)	
PAUSE statement	PAUSE 600	Deleted
ASSIGN statement and assigned GO TO statement	100 . . . ASSIGN 100 TO H . . . GO TO H . . .	Deleted
Assigned FORMAT specifiers	ASSIGN F TO 606	Deleted
H edit descriptors	606 FORMAT (9H1GOODBYE.)	Deleted
Computed GO TO statement	GO TO (10, 20, 30, 40), index	(obsolete)
Statement functions	FOIL(X, Y)= X**2 + 2*X*Y + Y**2	(obsolete)
DATA statements among executable statements	X= 27.3 DATA A, B, C / 5.0, 12.0. 13.0 / . . .	(obsolete)
CHARACTER* form of CHARACTER declaration	CHARACTER*8 STRING ! Use CHARACTER(8)	(obsolete)
Assumed character length functions	CHARACTER*(*) STRING	(obsolete) ^[14]
Fixed form source code	Column 1 contains *, ! or C for comments. Column 6 for continuation.	

"Hello world" example

```
program helloworld
    print *, "Hello, world."
end program helloworld
```

Fortran 95

Fortran 95, published officially as ISO/IEC 1539-1:1997, was a minor revision, mostly to resolve some outstanding issues from the Fortran 90 standard. Nevertheless, Fortran 95 also added a number of extensions, notably from the High Performance Fortran specification:

- FORALL and nested WHERE constructs to aid vectorization
- User-defined PURE and ELEMENTAL procedures
- Default initialization of derived type components, including pointer initialization
- Expanded the ability to use initialization expressions for data objects
- Clearly defined that ALLOCATABLE arrays are automatically deallocated when they go out of scope.

A number of intrinsic functions were extended (for example a `dim` argument was added to the `maxloc` intrinsic).

Several features noted in Fortran 90 to be "obsolete" were removed from Fortran 95:

- DO statements using REAL and DOUBLE PRECISION variables
- Branching to an END IF statement from outside its block
- PAUSE statement
- ASSIGN and assigned GOTO statement, and assigned format specifiers
- H edit descriptor.

An important supplement to Fortran 95 was the ISO technical report *TR-15581: Enhanced Data Type Facilities*, informally known as the *Allocatable TR*. This specification defined enhanced use of ALLOCATABLE arrays, prior to the availability of fully Fortran 2003-compliant Fortran compilers. Such uses include ALLOCATABLE arrays as derived type components, in procedure dummy argument lists, and as function return values. (ALLOCATABLE arrays are preferable to POINTER-based arrays because ALLOCATABLE arrays are guaranteed by Fortran 95 to be deallocated automatically when they go out of scope, eliminating the possibility of memory leakage. In addition, aliasing is not an issue for optimization of array references, allowing compilers to generate faster code than in the case of pointers.)

Another important supplement to Fortran 95 was the ISO technical report *TR-15580: Floating-point exception handling*, informally known as the *IEEE TR*. This specification defined support for IEEE floating-point arithmetic and floating point exception handling.

Conditional compilation and varying length strings

In addition to the mandatory "Base language" (defined in ISO/IEC 1539-1 : 1997), the Fortran 95 language also includes two optional modules:

- Varying character strings (ISO/IEC 1539-2 : 2000)
- Conditional compilation (ISO/IEC 1539-3 : 1998)

which, together, compose the multi-part International Standard (ISO/IEC 1539).

According to the standards developers, "the optional parts describe self-contained features which have been requested by a substantial body of users and/or implementors, but which are not deemed to be of sufficient generality for them to be required in all standard-conforming Fortran compilers." Nevertheless, if a standard-conforming Fortran does provide such options, then they "must be provided in accordance with the description of those facilities in the appropriate Part of the Standard."

Fortran 2003

Fortran 2003, officially published as ISO/IEC 1539-1:2004, is a major revision introducing many new features. A comprehensive summary of the new features of Fortran 2003 is available at the Fortran Working Group (ISO/IEC JTC1/SC22/WG5) official Web site.^[15]

From that article, the major enhancements for this revision include:

- Derived type enhancements: parameterized derived types, improved control of accessibility, improved structure constructors, and finalizers.
- Object-oriented programming support: type extension and inheritance, polymorphism, dynamic type allocation, and type-bound procedures, providing complete support for abstract data types.
- Data manipulation enhancements: allocatable components (incorporating TR 15581), deferred type parameters, VOLATILE attribute, explicit type specification in array constructors and allocate statements, pointer enhancements, extended initialization expressions, and enhanced intrinsic procedures.
- Input/output enhancements: asynchronous transfer, stream access, user specified transfer operations for derived types, user specified control of rounding during format conversions, named constants for preconnected units, the FLUSH statement, regularization of keywords, and access to error messages.
- Procedure pointers.
- Support for IEEE floating-point arithmetic and floating point exception handling (incorporating TR 15580).
- Interoperability with the C programming language.
- Support for international usage: access to ISO 10646 4-byte characters and choice of decimal or comma in numeric formatted input/output.
- Enhanced integration with the host operating system: access to command line arguments, environment variables, and processor error messages.

An important supplement to Fortran 2003 was the ISO technical report *TR-19767: Enhanced module facilities in Fortran*. This report provided *submodules*, which make Fortran modules more similar to Modula-2 modules. They are similar to Ada private child subunits. This allows the specification and implementation of a module to be expressed in separate program units, which improves packaging of large libraries, allows preservation of trade secrets while publishing definitive interfaces, and prevents compilation cascades.

Fortran 2008

The most recent standard, ISO/IEC 1539-1:2010, informally known as Fortran 2008, was approved in September 2010.^[16] As with Fortran 95, this is a minor upgrade, incorporating clarifications and corrections to Fortran 2003, as well as introducing a select few new capabilities. The new capabilities include:

- Submodules – Additional structuring facilities for modules; supersedes ISO/IEC TR 19767:2005
- Coarray Fortran – a parallel execution model
- The DO CONCURRENT construct – for loop iterations with no interdependencies
- The CONTIGUOUS attribute – to specify storage layout restrictions
- The BLOCK construct – can contain declarations of objects with construct scope
- Recursive allocatable components – as an alternative to recursive pointers in derived types

The Final Draft International Standard (FDIS) is available as document N1830.^[17]

An important supplement to Fortran 2008 is the ISO Technical Specification (TS) 29113 on *Further Interoperability of Fortran with C*,^{[18][19]} which has been submitted to ISO in May 2012 for approval. The specification adds support for accessing the array descriptor from C and allows to ignore the type and rank of arguments.

Fortran and supercomputers

Since Fortran has been in use for more than fifty years, there is a vast body of Fortran in daily use throughout the scientific and engineering communities. It is the primary language for some of the most intensive supercomputing tasks, such as weather and climate modeling, computational fluid dynamics, computational chemistry, computational economics, animal breeding, plant breeding and computational physics. Even today, half a century later, many of the floating-point benchmarks to gauge the performance of new computer processors are still written in Fortran (*e.g.*, CFP2006^[20], the floating-point component of the SPEC CPU2006^[21] benchmarks).

Portability

Portability was a problem in the early days because there was no agreed standard—not even IBM's reference manual—and computer companies vied to differentiate their offerings from others by providing incompatible features. Standards have improved portability. The 1966 standard provided a reference syntax and semantics, but vendors continued to provide incompatible extensions. Although careful programmers were coming to realize that use of incompatible extensions caused expensive portability problems, and were therefore using programs such as *The PFORT Verifier*, it was not until after the 1977 standard, when the National Bureau of Standards (now NIST) published *FIPS PUB 69*, that processors purchased by the U.S. Government were required to diagnose extensions of the standard. Rather than offer two processors, essentially every compiler eventually had at least an option to diagnose extensions.

Incompatible extensions were not the only portability problem. For numerical calculations, it is important to take account of the characteristics of the arithmetic. This was addressed by Fox et al. in the context of the 1966 standard by the *PORT* library. The ideas therein became widely used, and were eventually incorporated into the 1990 standard by way of intrinsic inquiry functions. The widespread (now almost universal) adoption of the IEEE 754 standard for binary floating-point arithmetic has essentially removed this problem.

Access to the computing environment (*e.g.* the program's command line, environment variables, textual explanation of error conditions) remained a problem until it was addressed by the 2003 standard.

Large collections of "library" software that could be described as being loosely related to engineering and scientific calculations, such as graphics libraries, have been written in C, and therefore access to them presented a portability problem. This has been addressed by incorporation of C interoperability into the 2003 standard.

It is now possible (and relatively easy) to write an entirely portable program in Fortran, even without recourse to a preprocessor.

Variants

Fortran 5

Fortran 5 was a programming language marketed by Data General Corp in the late 1970s and early 1980s, for the Nova, Eclipse, and MV line of computers. It had an optimizing compiler that was quite good for minicomputers of its time. The language most closely resembles Fortran 66. The name is a pun on the earlier Fortran IV.

Fortran V

Fortran V was a programming language distributed by Control Data Corporation in 1968 for the CDC 6600 series. The language was based upon Fortran IV.^[22]

Univac also offered a compiler for the 1100 series known as Fortran V. A spinoff of Univac Fortran V was Athena Fortran.

Fortran 6

Fortran 6 or Visual Fortran 2001 was licensed to Compaq by Microsoft. They have licensed Compaq Visual Fortran and have provided the Visual Studio 5 environment interface for Compaq v6 up to v6.1.^[23]

Specific variants

Vendors of high-performance scientific computers (*e.g.*, Burroughs, CDC, Cray, Honeywell, IBM, Texas Instruments, and UNIVAC) added extensions to Fortran to take advantage of special hardware features such as instruction cache, CPU pipelines, and vector arrays. For example, one of IBM's FORTRAN compilers (*H Extended IUP*) had a level of optimization which reordered the machine code instructions to keep multiple internal arithmetic units busy simultaneously. Another example is *CFD*, a special variant of Fortran designed specifically for the ILLIAC IV supercomputer, running at NASA's Ames Research Center. IBM Research Labs also developed an extended FORTRAN-based language called "VECTRAN" for processing of vectors and matrices.

Object-Oriented Fortran was an object-oriented extension of Fortran, in which data items can be grouped into objects, which can be instantiated and executed in parallel. It was available for Sun, Iris, iPSC, and nCUBE, but is no longer supported.

Such machine-specific extensions have either disappeared over time or have had elements incorporated into the main standards; the major remaining extension is OpenMP, which is a cross-platform extension for shared memory programming. One new extension, Coarray Fortran, is intended to support parallel programming.

FOR TRANSIT for the IBM 650

"FOR TRANSIT" was the name of a reduced version of the IBM 704 FORTRAN language, which was implemented for the IBM 650, using a translator program developed at Carnegie^[24] in the late 1950s. The following comment appears in the IBM Reference Manual ("FOR TRANSIT Automatic Coding System" C28-4038, Copyright 1957, 1959 by IBM):

The FORTRAN system was designed for a more complex machine than the 650, and consequently some of the 32 statements found in the FORTRAN Programmer's Reference Manual are not acceptable to the FOR TRANSIT system. In addition, certain restrictions to the FORTRAN language have been added. However, none of these restrictions make a source program written for FOR TRANSIT incompatible with the FORTRAN system for the 704.

The permissible statements were:

Arithmetic assignment statements, *e.g.* $a = b$

GO to n

GO TO (n_1, n_2, \dots, n_m), i

IF (a) n_1, n_2, n_3

PAUSE

STOP

DO n i = m1, m2

CONTINUE

END

READ n, list

PUNCH n, list

DIMENSION V, V, V, ...

EQUIVALENCE (a,b,c), (d,c), ...

Up to ten subroutines could be used in one program.

FOR TRANSIT statements were limited to columns 7 thru 56, only. Punched cards were used for input and output on the IBM 650. Three passes were required to translate source code to the "IT" language, then to compile the IT statements into SOAP assembly language, and finally to produce the object program, which could then be loaded into the machine to run the program (using punched cards for data input, and outputting results onto punched cards).

Two versions existed for the 650s with a 2000 word memory drum: FOR TRANSIT I (S) and FOR TRANSIT II, the latter for machines equipped with indexing registers and automatic floating point decimal (bi-quinary) arithmetic. Appendix A of the manual included wiring diagrams for the IBM 533 card reader/punch control panel.

Fortran-based languages

Prior to FORTRAN 77, a number of preprocessors were commonly used to provide a friendlier language, with the advantage that the preprocessed code could be compiled on any machine with a standard FORTRAN compiler. These preprocessors would typically support structured programming, variable names longer than six characters, additional data types, conditional compilation, and even macro capabilities. Popular preprocessors included FLECS, iftran, MORTRAN, SFtran, S-Fortran, Ratfor, and Ratfiv. Ratfor and Ratfiv, for example, implemented a C-like language, outputting preprocessed code in standard FORTRAN 66. Despite advances in the Fortran language, preprocessors continue to be used for conditional compilation and macro substitution.

LRLTRAN was developed at the Lawrence Radiation Laboratory to provide support for vector arithmetic and dynamic storage, among other extensions to support systems programming. The distribution included the LTSS operating system.

The Fortran-95 Standard includes an optional *Part 3* which defines an optional conditional compilation capability. This capability is often referred to as "CoCo".

Many Fortran compilers have integrated subsets of the C preprocessor into their systems.

SIMSCRIPT is an application specific Fortran preprocessor for modeling and simulating large discrete systems.

The F programming language was designed to be a clean subset of Fortran 95 that attempted to remove the redundant, unstructured, and deprecated features of Fortran, such as the EQUIVALENCE statement. F retains the array features added in Fortran 90, and removes control statements that were made obsolete by structured programming constructs added to both Fortran 77 and Fortran 90. F is described by its creators as "a compiled, structured, array programming language especially well suited to education and scientific computing."^[25]

Lahey and Fujitsu teamed up to create Fortran for the Microsoft .NET platform.^[26]

Code examples

The following program illustrates dynamic memory allocation and array-based operations, two features introduced with Fortran 90. Particularly noteworthy is the absence of DO loops and IF/THEN statements in manipulating the array; mathematical operations are applied to the array as a whole. Also apparent is the use of descriptive variable names and general code formatting that conform with contemporary programming style. This example computes an average over data entered interactively.

```
program average

! Read in some numbers and take the average
! As written, if there are no data points, an average of zero is
returned
! While this may not be desired behavior, it keeps this example
simple
```

```
implicit none

real, dimension(:), allocatable :: points
integer :: number_of_points
real :: average_points=0.,
positive_average=0., negative_average=0.

write (*,*) "Input number of points to average:"
read (*,*) number_of_points

allocate (points(number_of_points))

write (*,*) "Enter the points to average:"
read (*,*) points

! Take the average by summing points and dividing by number_of_points
if (number_of_points > 0) average_points = sum(points) /
number_of_points

! Now form average over positive and negative points only
if (count(points > 0.) > 0) then
    positive_average = sum(points, points > 0.) / count(points > 0.)
end if

if (count(points < 0.) > 0) then
    negative_average = sum(points, points < 0.) / count(points < 0.)
end if

deallocate (points)

! Print result to terminal
write (*,'(a,g12.4)') 'Average = ', average_points
write (*,'(a,g12.4)') 'Average of positive points = ',
positive_average
write (*,'(a,g12.4)') 'Average of negative points = ',
negative_average

end program average
```

Humor

During the same Fortran Standards Committee meeting at which the name "FORTRAN 77" was chosen, a technical proposal was incorporated into the official distribution, bearing the title, "Letter O considered harmful". This proposal purported to address the confusion that sometimes arises between the letter "O" and the numeral zero, by eliminating the letter from allowable variable names. However, the method proposed was to eliminate the letter from the character set entirely (thereby retaining 48 as the number of lexical characters, which the colon had increased to 49). This was considered beneficial in that it would promote structured programming, by making it impossible to use

the notorious **GO TO** statement as before. (Troublesome **FORMAT** statements would also be eliminated.) It was noted that this "might invalidate some existing programs" but that most of these "probably were non-conforming, anyway".^{[27][28]}

During the standards committee battle over whether the "minimum trip count" for the FORTRAN 77 **DO** statement should be zero (allowing no execution of the block) or one (the "plunge-ahead" **DO**), another facetious alternative was proposed (by Loren Meissner) to have the minimum trip be two—since there is no need for a loop if it is only executed once.

References

- [1] "Math 169 Notes - Santa Clara University" (<http://math.scu.edu/~dsmolars/ma169/notesfortran.html>). .
- [2] Eugene Loh (18 June 2010). "The Ideal HPC Programming Language" (<http://queue.acm.org/detail.cfm?id=1820518>). *Queue* (Association of Computing Machines) **8** (6). .
- [3] Softwarepreservation.org (http://www.softwarepreservation.org/projects/FORTRAN/index.html#By_FORTRAN_project_members)
- [4] Mindell, David, DIGITAL APOLLO, MIT Press, Cambridge MA, 2008, p.99
- [5] The Fortran I Compiler (<http://polaris.cs.uiuc.edu/publications/c1070.pdf>) "The Fortran I compiler was the first major project in code optimization. It tackled problems of crucial importance whose general solution was an important research focus in compiler technology for several decades. Many classical techniques for compiler analysis and optimization can trace their origins and inspiration to the Fortran I compiler."
- [6] Fortran creator John Backus dies - Gadgets - MSNBC.com (<http://www.msnbc.msn.com/id/17704662/>), MSNBC.com
- [7] Bitsavers.org (http://www.bitsavers.org/pdf/ibm/fortran/F28-8074-3_FORTRANII_GenInf.pdf)
- [8] Ibiblio.org (<http://www.ibiblio.org/pub/languages/fortran/ch1-12.html>)
- [9] Haines, L. H. (1965). "Serial compilation and the 1401 FORTRAN compiler" (<http://domino.research.IBM.com/tchjr/journalindex.nsf/495f80c9d0f539778525681e00724804/cde711e5ad6786e485256bfa00685a03?OpenDocument>). *IBM Systems Journal* **4** (1): 73–80. doi:10.1147/sj.41.0073. . This article was reprinted, edited, in both editions of Lee, John A. N. (1967(1st), 1974(2nd)). *Anatomy of a Compiler*. Van Nostrand Reinhold.
- [10] McCracken, Daniel D. (1965). "Preface". *A Guide to FORTRAN IV Programming*. New York: Wiley. p. v. ISBN 0-471-58281-6.
- [11] Chilton Computing with FORTRAN (<http://www.chilton-computing.org.uk/acd/literature/reports/p008.htm>), Chilton-computing.org.uk
- [12] Mil-std-1753. *DoD Supplement to X3.9-1978* (http://www.fortran.com/fortran/mil_std_1753.html). United States Government Printing Office. .
- [13] Posix 1003.9-1992. *POSIX FORTRAN 77 Language Interface – Part 1: Binding for System Application Program Interface API* (http://standards.ieee.org/reading/ieee/std_public/description posix/1003.9-1992_desc.html). IEEE. .
- [14] "Fortran Variable Declarations" (<http://h21007.www2.hp.com/portal/download/files/unprot/fortran/docs/lrm/lrm0085.htm>) (in English) (html). *Compaq Fortran*. Texas, Huston, US: Compaq Computer Corporation. 1999. . Retrieved 14 സെപ്റ്റംബർ 2012. "The form CHARACTER*(*) is an obsolescent feature in Fortran 95."
- [15] Fortran Working Group (WG5) (<http://www.nag.co.uk/sc22wg5/>). It may also be downloaded as a PDF file (<ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.pdf>) or zipped PostScript file (<ftp://ftp.nag.co.uk/sc22wg5/N1551-N1600/N1579.ps.gz>), FTP.nag.co.uk
- [16] N1836, Summary of Voting/Table of Replies on ISO/IEC FDIS 1539-1, Information technology - Programming languages - Fortran - Part 1: Base language <ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1836.pdf> PDF (101 KiB)
- [17] N1830, Information technology — Programming languages — Fortran — Part 1: Base language <ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf> PDF (7.9 MiB)
- [18] ISO page to ISO/IEC DTS 29113, Further Interoperability of Fortran with C (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=45136)
- [19] Draft of the Technical Specification (TS) 29113 <ftp://ftp.nag.co.uk/sc22wg5/N1901-N1950/N1917.pdf> PDF (312 kiB)
- [20] <http://www.spec.org/cpu2006/CFP2006/>
- [21] <http://www.spec.org/cpu2006/>
- [22] Healy, MJR (1968). "Towards FORTRAN VI" (http://www.cs-software.com/software/fortran/compaq/cvf_relnotes.html#61ver_news). 15 March 2011. .
- [24] "Internal Translator (IT) A Compiler for the IBM 650", by A. J. Perlis, J. W. Smith, and H. R. Van Zoeren, Computation Center, Carnegie Institute of Technology
- [25] "F Programming Language Homepage" (<http://www.fortran.com/F/index.html>). .
- [26] "Fortran for .NET Language System" (<http://www.lahey.com/lf71/lfnets.htm>). .
- [27] X3J3 post-meeting distribution for meeting held at Brookhaven National Laboratory in November 1976.

[28] "The obliteration of O", Computer Weekly, 3 March 1977

Further reading

Articles

- Allen, F.E. (September 1981). "A History of Language Processor Technology in IBM" (<http://www.research.ibm.com/journal/rd/255/ibmrd2505Q.pdf>). *IBM Journal of Research and Development* (IBM) **25** (5).
- Backus, J. W.; H. Stern, I. Ziller, R. A. Hughes, R. Nutt, R. J. Beeber, S. Best, R. Goldberg, L. M. Haibt, H. L. Herrick, R. A. Nelson, D. Sayre, P. B. Sheridan (1957). "The FORTRAN Automatic Coding System". *Western joint computer conference: Techniques for reliability* (Los Angeles, California: Institute of Radio Engineers, American Institute of Electrical Engineers, ACM): 188–198. doi:10.1145/1455567.1455599.
- Chivers, Ian D.; Sleightholme, Jane (2012). "Compiler support for the Fortran 2003 & 2008 standards" (http://www.fortranplus.co.uk/resources/compiler_table.pdf). *ACM SIGPLAN Fortran Forum* (ACM) **28** (1): 26–28. doi:10.1145/1520752.1520755. ISSN 10617264.
- Pigott, Diarmuid (2006). "FORTRAN - Backus et al high-level compiler (Computer Language)" (<http://hopl.murdoch.edu.au/showlanguage.prx?exp=8&language=FORTRAN>). *The Encyclopedia of Computer Languages*. Murdoch University. Retrieved 5 May 2010.
- Roberts, Mark L.; Griffiths, Peter D. (1985). "Design Considerations for IBM Personal Computer Professional FORTRAN, an Optimizing Compiler" (<http://www.research.ibm.com/journal/sj/241/ibmsj2401G.pdf>). *IBM Systems Journal* (IBM) **24** (1): 49–60. doi:10.1147/sj.241.0049.

"Core" language standards

- Ansi x3.9-1966. *USA Standard FORTRAN* (<http://www.fh-jena.de/~kleine/history/languages/ansi-x3dot9-1966-Fortran66.pdf>). American National Standards Institute. Informally known as FORTRAN 66.
- Ansi x3.9-1978. *American National Standard – Programming Language FORTRAN* (http://www.fortran.com/fortran/F77_std/rjcnf.html). American National Standards Institute. Also known as ISO 1539-1980, informally known as FORTRAN 77.
- ANSI X3.198-1992 (R1997) / ISO/IEC 1539:1991. *American National Standard – Programming Language Fortran Extended* (<http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=17366>). American National Standards Institute / ISO/IEC. Informally known as Fortran 90.
- ISO/IEC 1539-1:1997. *Information technology – Programming languages – Fortran – Part 1: Base language* (<http://j3-fortran.org/doc/standing/archive/007/97-007r2/pdf/97-007r2.pdf>). Informally known as Fortran 95. There are a further two parts to this standard. Part 1 has been formally adopted by ANSI.
- ISO/IEC 1539-1:2004. *Information technology – Programming languages – Fortran – Part 1: Base language* (<http://www.dkuug.dk/jtc1/sc22/open/n3661.pdf>). Informally known as Fortran 2003.
- ISO/IEC 1539-1:2010 (Final Draft International Standard). *Information technology – Programming languages – Fortran – Part 1: Base language* (<ftp://ftp.nag.co.uk/sc22wg5/N1801-N1850/N1830.pdf>). Informally known as Fortran 2008.

Related standards

- Kneis, Wilfried (October 1981). "Draft standard Industrial Real-Time FORTRAN". *ACM SIGPLAN Notices* (ACM Press) **16** (7): 45–60. doi:10.1145/947864.947868. ISSN 0362-1340.
- *ISO 8651-1:1988 Information processing systems—Computer graphics—Graphical Kernel System (GKS) language bindings—Part 1: FORTRAN* (http://www.iso.org/iso/catalogue_detail?csnumber=16024). Geneva, Switzerland: ISO. 1988.

Textbooks

- Adams, Jeanne C.; Brainerd, Walter S.; Hendrickson, Richard A.; Maine, Richard E.; Martin, Jeanne T.; Smith, Brian T. (2009). *The Fortran 2003 Handbook* (1st ed.). Springer. ISBN 978-1-84628-378-9.

- Akin, Ed (2003). *Object Oriented Programming via Fortran 90/95* (1st ed.). Cambridge University Press. ISBN 0-521-52408-3.
- Chapman, Stephen J. (2007). *Fortran 95/2003 for Scientists and Engineers* (3rd ed.). McGraw-Hill. ISBN 978-0-07-319157-7.
- Chivers, Ian; Sleighholme, Jane (2012). *Introduction to Programming with Fortran* (2nd ed.). Springer. ISBN 978-0-85729-232-2.
- Etter, D. M. (1990). *Structured FORTRAN 77 for Engineers and Scientists* (3rd ed.). The Benjamin/Cummings Publishing Company, Inc.. ISBN 0-8053-0051-1.
- Ellis, T. M. R.; Phillips, Ivor R.; Lahey, Thomas M. (1994). *Fortran 90 Programming* (1st ed.). Addison Wesley. ISBN 0-201-54446-6.
- Kupferschmid, Michael (2002). *Classical Fortran: Programming for Engineering and Scientific Applications*. Marcel Dekker (CRC Press). ISBN 0-8247-0802-4.
- McCracken, Daniel D. (1961). *A Guide to FORTRAN Programming*. New York: Wiley. LCCN 61016618.
- Metcalf, Michael; John Reid, Malcolm Cohen (2011). *Modern Fortran Explained*. Oxford University Press. ISBN 0-19-960142-9.
- Nyhoff, Larry; Sanford Leestma (1995). *FORTRAN 77 for Engineers and Scientists with an Introduction to Fortran 90* (4th ed.). Prentice Hall. ISBN 0-13-363003-X.
- Page, Clive G. (1988). *Professional Programmer's Guide to Fortran77* (<http://www.star.le.ac.uk/~cgp/prof77.html>) (7 June 2005 ed.). London: Pitman. ISBN 0-273-02856-1. Retrieved 4 May 2010.
- Press, William H. (1996). *Numerical Recipes in Fortran 90: The Art of Parallel Scientific Computing* (<http://www.nrbook.com/a/bookf90pdf.php>). Cambridge, UK: Cambridge University Press. ISBN 0-521-57439-0.
- Sleighhome, Jane; Chivers, Ian David (1990). *Interactive Fortran 77: A Hands-On Approach* (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.95.9503>). Computers and their applications (2nd ed.). Chichester: E. Horwood. ISBN 0-13-466764-6.

External links

- JTC1/SC22/WG5 (<http://www.nag.co.uk/sc22wg5/>) — The ISO/IEC Fortran Working Group
- History of Fortran (<http://www.softwarepreservation.org/projects/FORTRAN/>) at the Computer History Museum's Software Preservation Group
- Bemer, Bob, "Who Was Who in IBM's Programming Research? -- Early FORTRAN Days" (<http://www.trailing-edge.com/~bobbemer/PRORES.HTM>), January 1957, Computer History Vignettes
- Comprehensive Fortran Standards Documents (<http://gcc.gnu.org/wiki/GFortranStandards>) from the gfortran project
- Fortran IV - IBM System/360 and System/370 Fortran IV Language - GC28-6515-10 (<http://www.fh-jena.de/~kleine/history/languages/GC28-6515-10-FORTRAN-IV-Language.pdf>)
- Fortran 77, 90, 95, 2003, 2008 Information & Resources (http://www.fortranplus.co.uk/fortran_info.html)
- Fortran 77 4.0 Reference Manual (http://www.physics.ucdavis.edu/~vem/F77_Ref.pdf) (This link is broken)
- Fortran 90 Reference Card (http://www.pa.msu.edu/~duxbury/courses/phy480/fortran90_refcard.pdf)
- ECMA-9 Fortran Standard (1965) (<http://www.ecma-international.org/publications/files/ECMA-ST-WITHDRAWN/ECMA-9, 1st Edition, April 1965.pdf>)
- Online FORTRAN compiler for small experiments (<http://www.vintagebigblue.org/Compilerator/FORTRAN/forthCompile.php>)

From Math to Programming

Lambda calculus

Lambda calculus (also written as λ -calculus or called "the lambda calculus") is a formal system in mathematical logic for expressing computation by way of variable binding and substitution. It was first formulated by Alonzo Church as a way to formalize mathematics through the notion of functions, in contrast to the field of set theory. Although not very successful in that respect, the lambda calculus found early successes in the area of computability theory, such as a negative answer to Hilbert's Entscheidungsproblem.

Because of the importance of the notion of variable binding and substitution, there is not just one system of lambda calculus. Historically, the most important system was the untyped lambda calculus. In the untyped lambda calculus, function application has no restrictions (so the notion of the domain of a function is not built into the system). In the Church–Turing Thesis, the untyped lambda calculus is claimed to be capable of computing all effectively calculable functions. The typed lambda calculus is a variety that restricts function application, so that functions can only be applied if they are capable of accepting the given input's "type" of data.

Today, the lambda calculus has applications in many different areas in mathematics, philosophy, and computer science. It is still used in the area of computability theory, although Turing machines are arguably the preferred model for computation. Lambda calculus has played an important role in the development of the theory of programming languages. The most prominent counterparts to lambda calculus in computer science are functional programming languages, which essentially implement the calculus (augmented with some constants and datatypes). Beyond programming languages, the lambda calculus also has many applications in proof theory. A major example of this is the Curry–Howard correspondence, which gives a correspondence between different systems of typed lambda calculus and systems of formal logic.

Lambda calculus in history of mathematics

The lambda calculus was introduced by mathematician Alonzo Church in the 1930s as part of an investigation into the foundations of mathematics.^{[1][2]} The original system was shown to be logically inconsistent in 1935 when Stephen Kleene and J. B. Rosser developed the Kleene–Rosser paradox.

Subsequently, in 1936 Church isolated and published just the portion relevant to computation, what is now called the untyped lambda calculus.^[3] In 1940, he also introduced a computationally weaker, but logically consistent system, known as the simply typed lambda calculus.^[4]

Informal description

Motivation

Recursive functions are a fundamental concept within computer science and mathematics. The λ -calculus provides simple semantics for computation, enabling properties of computation to be studied formally.

Consider the following two examples. The identity function

$$\text{id}(x) = x$$

takes a single input, x , and immediately returns x (i.e. the identity does nothing with its input), whereas the function

$$\text{sqsum}(x, y) = x \times x + y \times y$$

takes a pair of inputs, x and y and returns the sum of their squares, $x \times x + y \times y$. Using these two examples, we can make some useful observations that motivate the major ideas in lambda calculus.

The first observation is that functions need not be explicitly named. That is, the function

$$\text{sqsum}(x, y) = x \times x + y \times y$$

can be rewritten in *anonymous form* as

$$(x, y) \mapsto x \times x + y \times y$$

(read as “the pair of x and y is mapped to $x \times x + y \times y$ ”). Similarly,

$$\text{id}(x) = x$$

can be rewritten in anonymous form as $x \mapsto x$, where the input is simply mapped to itself.

The second observation is that the specific choice of name for a function's arguments is largely irrelevant. That is,

$$x \mapsto x \text{ and}$$

$$y \mapsto y$$

express the same function: the identity. Similarly,

$$(x, y) \mapsto x \times x + y \times y \text{ and}$$

$$(u, v) \mapsto u \times u + v \times v$$

also express the same function.

Finally, any function that requires two inputs, for instance the aforementioned `sqsum` function, can be reworked into an equivalent function that accepts a single input, and as output returns *another* function, that in turn accepts a single input. For example,

$$(x, y) \mapsto x \times x + y \times y$$

can be reworked into

$$x \mapsto (y \mapsto x \times x + y \times y)$$

This transformation is called currying, i.e. transforming a function that takes multiple arguments in such a way that it can be called as a *chain of functions each with a single argument* (partial application). It can be generalized to functions accepting an arbitrary number of arguments.

Applying the above function to the arguments $(5, 2)$, we have:

$$\begin{aligned} & ((x, y) \mapsto x \times x + y \times y)(5, 2) \\ &= 5 \times 5 + 2 \times 2 = 29 \end{aligned}$$

However, using currying, we have:

$$\begin{aligned} & (x \mapsto (y \mapsto x \times x + y \times y))(5) \\ &= (y \mapsto 5 \times 5 + y \times 5)(2) \end{aligned}$$

$$= 5 \times 5 + 2 \times 2 = 29$$

and we see the uncurried and curried forms compute the same result. Notice that $x \times x$ became a constant after the first argument assignment.

The lambda calculus

The lambda calculus consists of a language of **lambda terms** along with an equational theory (which can also be understood operationally).

Since the names of functions are largely a convenience, the lambda calculus has no means of naming a function. Furthermore, since all functions accepting more than one input can be transformed into equivalent functions accepting a single input (via Currying), the lambda calculus has no means for creating a function that accepts more than one argument. Finally, since the names of arguments are largely irrelevant, the naive notion of equality on lambda terms is **alpha-equivalence** (see below), which codifies this principle.

Lambda terms

The syntax of lambda terms is particularly simple. There are three ways in which to obtain them:

- a lambda term may be a variable, x ;
- if t is a lambda term, and x is a variable, then $\lambda x.t$ is a lambda term (called a **lambda abstraction**);
- if t and s are lambda terms, then ts is a lambda term (called an **application**).

Nothing else is a lambda term, though bracketing may be used and may be needed to disambiguate terms. For example, $\lambda x.((\lambda x.x)x)$ and $(\lambda x.(\lambda x.x))x$ denote different terms.

Intuitively, a lambda abstraction $\lambda x.t$ represents an anonymous function that takes a single input, and the λ is said to **bind** x in t , and an application ts represents the application of input s to some function t . In the lambda calculus, functions are taken to be 'first class values', so functions may be used as the inputs to other functions, and functions may return functions as their outputs.

For example, $\lambda x.x$ represents the identity function, $x \rightarrow x$, and $(\lambda x.x)y$ represents the identity function applied to y . Further, $(\lambda x.y)$ represents the **constant function** $x \rightarrow y$, the function that always returns y , no matter the input. It should be noted that in lambda calculus, function application is regarded as left-associative, so that stx means $(st)x$. Lambda terms on their own aren't particularly interesting. What makes them interesting are the various notions of 'equivalence' and 'reduction' that can be defined over them.

Alpha equivalence

A basic form of equivalence, definable on lambda terms, is alpha equivalence. It captures the intuition that the particular choice of a bound variable, in a lambda abstraction, doesn't (usually) matter. For instance, $\lambda x.x$ and $\lambda y.y$ are alpha-equivalent lambda terms, representing the same identity function. Note that the terms x and y aren't alpha-equivalent, because they are not bound in a lambda abstraction. In many presentations, it is usual to identify alpha-equivalent lambda terms.

The following definitions are necessary in order to be able to define beta reduction.

Free variables

The **free variables** of a term are those variables not bound by a lambda abstraction. That is, the free variables of x are just x ; the free variables of $\lambda x. t$ are the free variables of t , with x removed, and the free variables of $t s$ are the union of the free variables of t and s .

For example, the lambda term representing the identity $\lambda x. x$ has no free variables, but the constant function $\lambda x. y$ has a single free variable, y .

Capture-avoiding substitutions

Using the definition of free variables, we may now define a capture-avoiding substitution. Suppose t , s and r are lambda terms and x and y are variables. We write $t[x := r]$ for the substitution of r for x in t , in a capture-avoiding manner. That is:

- $x[x := r] = r$;
- $y[x := r] = y$ if $x \neq y$;
- $(ts)[x := r] = (t[x := r])(s[x := r])$;
- $(\lambda x.t)[x := r] = \lambda x.t$;
- $(\lambda y.t)[x := r] = \lambda y.(t[x := r])$ if $x \neq y$ and y is not in the free variables of r (sometimes said y is **fresh for r**).

For example, $(\lambda x.x)[y := y] = \lambda x.(x[y := y]) = \lambda x.x$, and

$$((\lambda x.y)x)[x := y] = ((\lambda x.y)[x := y])(x[x := y]) = (\lambda x.y)y.$$

The freshness condition (requiring that y is not in the free variables of r) is crucial in order to ensure that substitution does not change the meaning of functions. For example, suppose we define another substitution action without the freshness condition. Then, $(\lambda x.y)[y := x] = \lambda x.(y[y := x]) = \lambda x.x$, and the constant function $\lambda x.y$ turns into the identity $\lambda x.x$ by substitution.

If our freshness condition is not met, then we may simply alpha-rename with a suitably fresh variable. For example, switching back to our correct notion of substitution, in $(\lambda y.x)[x := y]$ the lambda abstraction can be renamed with a fresh variable z , to obtain $(\lambda z.x)[x := y] = \lambda z.(x[x := y]) = \lambda z.y$, and the meaning of the function is preserved by substitution.

Beta reduction

Beta reduction states that an application of the form $(\lambda x.t)s$ reduces to the term $t[x := s]$ (we write $(\lambda x.t)s \rightarrow t[x := s]$ as a convenient shorthand for " $(\lambda x.t)s$ beta reduces to $t[x := s]$ "). For example, for every s we have $(\lambda x.x)s \rightarrow x[x := s] = s$, demonstrating that $\lambda x.x$ really is the identity. Similarly, $(\lambda x.y)s \rightarrow y[x := s] = y$, demonstrating that $\lambda x.y$ really is a constant function.

The lambda calculus may be seen as an idealised functional programming language, like Haskell or Standard ML. Under this view, beta reduction corresponds to a computational step, and in the untyped lambda calculus, as presented here, reduction need not terminate. For instance, consider the term $(\lambda x.xx)(\lambda x.xx)$. Here, we have $(\lambda x.xx)(\lambda x.xx) \rightarrow (xx)[x := \lambda x.xx] = (x[x := \lambda x.xx])(x[x := \lambda x.xx]) = (\lambda x.xx)(\lambda x.xx)$. That is, the term reduces to itself in a single beta reduction, and therefore reduction will never terminate.

Another problem with the untyped lambda calculus is the inability to distinguish between different kinds of data. For instance, we may want to write a function that only operates on numbers. However, in the untyped lambda calculus, there's no way to prevent our function from being applied to truth values, or strings, for instance.

Formal definition

Definition

Lambda expressions are composed of

- variables $v_1, v_2, \dots, v_n, \dots$
- the abstraction symbols λ and $.$
- parentheses $()$

The set of lambda expressions, Λ , can be defined inductively:

1. If x is a variable, then $x \in \Lambda$
2. If x is a variable and $M \in \Lambda$, then $(\lambda x. M) \in \Lambda$
3. If $M, N \in \Lambda$, then $(M N) \in \Lambda$

Instances of rule 2 are known as abstractions and instances of rule 3 are known as applications.^[5]

Notation

To keep the notation of lambda expressions uncluttered, the following conventions are usually applied.

- Outermost parentheses are dropped: $M N$ instead of $(M N)$
- Applications are assumed to be left associative: $M N P$ may be written instead of $((M N) P)$ ^[6]
- The body of an abstraction extends as far right as possible: $\lambda x. M N$ means $\lambda x. (M N)$ and not $(\lambda x. M) N$
- A sequence of abstractions is contracted: $\lambda x. \lambda y. \lambda z. N$ is abbreviated as $\lambda xyz. N$ ^{[7][8]}

Free and bound variables

The abstraction operator, λ , is said to bind its variable wherever it occurs in the body of the abstraction. Variables that fall within the scope of a lambda are said to be *bound*. All other variables are called *free*. For example in the following expression y is a bound variable and x is free: $\lambda y. x x y$. Also note that a variable binds to its "nearest" lambda. In the following expression one single occurrence of x is bound by the second lambda: $\lambda x. y (\lambda x. z x)$

The set of *free variables* of a lambda expression, M , is denoted as $FV(M)$ and is defined by recursion on the structure of the terms, as follows:

1. $FV(x) = \{x\}$, where x is a variable
2. $FV(\lambda x. M) = FV(M) \setminus \{x\}$
3. $FV(M N) = FV(M) \cup FV(N)$ ^[9]

An expression that contains no free variables is said to be *closed*. Closed lambda expressions are also known as combinators and are equivalent to terms in combinatory logic.

Reduction

The meaning of lambda expressions is defined by how expressions can be reduced.^[10]

There are three kinds of reduction:

- **α -conversion**: changing bound variables;
- **β -reduction**: applying functions to their arguments;
- **η -conversion**: which captures a notion of extensionality.

We also speak of the resulting equivalences: two expressions are β -equivalent, if they can be β -converted into the same expression, and α/η -equivalence are defined similarly.

The term *redex*, short for *reducible expression*, refers to subterms that can be reduced by one of the reduction rules. For example, $(\lambda x. M) N$ is a beta-redex; if x is not free in M , $\lambda x. M x$ is an eta-redex. The expression to which a

redex reduces is called its reduct; using the previous example, the reducts of these expressions are respectively $M[x := N]$ and M .

α -conversion

Alpha-conversion, sometimes known as alpha-renaming,^[11] allows bound variable names to be changed. For example, alpha-conversion of $\lambda x.x$ might yield $\lambda y.y$. Terms that differ only by alpha-conversion are called *α -equivalent*. Frequently in uses of lambda calculus, α -equivalent terms are considered to be equivalent.

The precise rules for alpha-conversion are not completely trivial. First, when alpha-converting an abstraction, the only variable occurrences that are renamed are those that are bound to the same abstraction. For example, an alpha-conversion of $\lambda x.\lambda x.x$ could result in $\lambda y.\lambda x.x$, but it could *not* result in $\lambda y.\lambda x.y$. The latter has a different meaning from the original.

Second, alpha-conversion is not possible if it would result in a variable getting captured by a different abstraction. For example, if we replace x with y in $\lambda x.\lambda y.x$, we get $\lambda y.\lambda y.y$, which is not at all the same.

In programming languages with static scope, alpha-conversion can be used to make name resolution simpler by ensuring that no variable name masks a name in a containing scope (see alpha renaming to make name resolution trivial).

Substitution

Substitution, written $E[V := E']$, is the process of replacing all free occurrences of the variable V by expression E' . Substitution on terms of the λ -calculus is defined by recursion on the structure of terms, as follows.

$$\begin{aligned} x[x := N] &\equiv N \\ y[x := N] &\equiv y, \text{ if } x \neq y \\ (M_1 M_2)[x := N] &\equiv (M_1[x := N]) (M_2[x := N]) \\ (\lambda x.M)[x := N] &\equiv \lambda x.M \\ (\lambda y.M)[x := N] &\equiv \lambda y.(M[x := N]), \text{ if } x \neq y, \text{ provided } y \notin FV(N) \end{aligned}$$

To substitute into a lambda abstraction, it is sometimes necessary to α -convert the expression. For example, it is not correct for $(\lambda x.y)[y := x]$ to result in $(\lambda x.x)$, because the substituted x was supposed to be free but ended up being bound. The correct substitution in this case is $(\lambda z.z)$, up to α -equivalence. Notice that substitution is defined uniquely up to α -equivalence.

β -reduction

Beta-reduction captures the idea of function application. Beta-reduction is defined in terms of substitution: the beta-reduction of $((\lambda V.E) E')$ is $E[V := E']$.

For example, assuming some encoding of 2 , 7 , \times , we have the following β -reductions: $((\lambda n.n \times 2) 7) \rightarrow 7 \times 2$.

η -conversion

Eta-conversion expresses the idea of extensionality, which in this context is that two functions are the same if and only if they give the same result for all arguments. Eta-conversion converts between $\lambda x.(f x)$ and f whenever x does not appear free in f .

Normal forms and confluence

For the untyped lambda calculus, β -reduction as a rewriting rule is neither strongly normalising nor weakly normalising.

However, it can be shown that β -reduction is confluent. (Of course, we are working up to α -conversion, i.e. we consider two normal forms to be equal, if it is possible to α -convert one into the other.)

Therefore, both strongly normalising terms and weakly normalising terms have a unique normal form. For strongly normalising terms, any reduction strategy is guaranteed to yield the normal form, whereas for weakly normalising terms, some reduction strategies may fail to find it.

Encoding datatypes

The basic lambda calculus may be used to model booleans, arithmetic, data structures and recursion, as illustrated in the following sub-sections.

Arithmetic in lambda calculus

There are several possible ways to define the natural numbers in lambda calculus, but by far the most common are the Church numerals, which can be defined as follows:

$$\begin{aligned} 0 &:= \lambda f. \lambda x. x \\ 1 &:= \lambda f. \lambda x. f x \\ 2 &:= \lambda f. \lambda x. f (f x) \\ 3 &:= \lambda f. \lambda x. f (f (f x)) \end{aligned}$$

and so on. Or using the alternate syntax presented above in *Notation*:

$$\begin{aligned} 0 &:= \lambda f x. x \\ 1 &:= \lambda f x. f x \\ 2 &:= \lambda f x. f (f x) \\ 3 &:= \lambda f x. f (f (f x)) \end{aligned}$$

A Church numeral is a higher-order function—it takes a single-argument function f , and returns another single-argument function. The Church numeral n is a function that takes a function f as argument and returns the n -th composition of f , i.e. the function f composed with itself n times. This is denoted $f^{(n)}$ and is in fact the n -th power of f (considered as an operator); $f^{(0)}$ is defined to be the identity function. Such repeated compositions (of a single function f) obey the laws of exponents, which is why these numerals can be used for arithmetic. (In Church's original lambda calculus, the formal parameter of a lambda expression was required to occur at least once in the function body, which made the above definition of 0 impossible.)

We can define a successor function, which takes a number n and returns $n + 1$ by adding an additional application of f :

$$\text{SUCC} := \lambda n. \lambda f. \lambda x. f (n f x)$$

Because the m -th composition of f composed with the n -th composition of f gives the $m+n$ -th composition of f , addition can be defined as follows:

$$\text{PLUS} := \lambda m. \lambda n. \lambda f. \lambda x. m f (n f x)$$

`PLUS` can be thought of as a function taking two natural numbers as arguments and returning a natural number; it can be verified that

$$\text{PLUS } 2 \ 3$$

and

5

are equivalent lambda expressions. Since adding m to a number n can be accomplished by adding 1 m times, an equivalent definition is:

$$\text{PLUS} := \lambda m. \lambda n. m \text{ SUCC } n^{[12]}$$

Similarly, multiplication can be defined as

$$\text{MULT} := \lambda m. \lambda n. \lambda f. m (n f)^{[13]}$$

Alternatively

$$\text{MULT} := \lambda m. \lambda n. m (\text{PLUS } n) 0$$

since multiplying m and n is the same as repeating the add n function m times and then applying it to zero. Exponentiation has a rather simple rendering in Church numerals, namely

$$\text{POW} := \lambda b. \lambda e. e b$$

The predecessor function defined by $\text{PRED } n = n - 1$ for a positive integer n and $\text{PRED } 0 = 0$ is considerably more difficult. The formula

$$\text{PRED} := \lambda n. \lambda f. \lambda x. n (\lambda g. \lambda h. h (g f)) (\lambda u. x) (\lambda u. u)$$

can be validated by showing inductively that if T denotes $(\lambda g. \lambda h. h (g f))$, then $T^{(n)} (\lambda u. x) = (\lambda h. h (f^{(n-1)} (x)))$ for $n > 0$. Two other definitions of PRED are given below, one using conditionals and the other using pairs. With the predecessor function, subtraction is straightforward. Defining

$$\text{SUB} := \lambda m. \lambda n. n \text{ PRED } m,$$

$\text{SUB } m n$ yields $m - n$ when $m > n$ and 0 otherwise.

Logic and predicates

By convention, the following two definitions (known as Church booleans) are used for the boolean values TRUE and FALSE:

$$\text{TRUE} := \lambda x. \lambda y. x$$

$$\text{FALSE} := \lambda x. \lambda y. y$$

(Note that FALSE is equivalent to the Church numeral zero defined above)

Then, with these two λ -terms, we can define some logic operators (these are just possible formulations; other expressions are equally correct):

$$\text{AND} := \lambda p. \lambda q. p q p$$

$$\text{OR} := \lambda p. \lambda q. p p q$$

$$\text{NOT} := \lambda p. \lambda a. \lambda b. p b a$$

$$\text{IFTHENELSE} := \lambda p. \lambda a. \lambda b. p a b$$

We are now able to compute some logic functions, for example:

$$\text{AND } \text{TRUE } \text{FALSE}$$

$$\equiv (\lambda p. \lambda q. p q p) \text{ TRUE } \text{FALSE} \rightarrow_{\beta} \text{TRUE } \text{FALSE } \text{TRUE}$$

$$\equiv (\lambda x. \lambda y. x) \text{ FALSE } \text{TRUE} \rightarrow_{\beta} \text{FALSE}$$

and we see that AND TRUE FALSE is equivalent to FALSE.

A *predicate* is a function that returns a boolean value. The most fundamental predicate is ISZERO, which returns TRUE if its argument is the Church numeral 0, and FALSE if its argument is any other Church numeral:

$$\text{ISZERO} := \lambda n. n (\lambda x. \text{FALSE}) \text{ TRUE}$$

The following predicate tests whether the first argument is less-than-or-equal-to the second:

```
LEQ := λm.λn.ISZERO (SUB m n),
```

and since $m = n$, if $\text{LEQ } m n$ and $\text{LEQ } n m$, it is straightforward to build a predicate for numerical equality.

The availability of predicates and the above definition of TRUE and FALSE make it convenient to write "if-then-else" expressions in lambda calculus. For example, the predecessor function can be defined as:

```
PRED := λn.n (λg.λk.ISZERO (g 1) k (PLUS (g k) 1)) (λv.0) 0
```

which can be verified by showing inductively that $n (\lambda g. \lambda k. \text{ISZERO } (g 1) k (\text{PLUS } (g k) 1)) (\lambda v. 0)$ is the add $n - 1$ function for $n > 0$.

Pairs

A pair (2-tuple) can be defined in terms of TRUE and FALSE, by using the Church encoding for pairs. For example, PAIR encapsulates the pair (x, y) , FIRST returns the first element of the pair, and SECOND returns the second.

```
PAIR := λx.λy.λf.f x y
```

```
FIRST := λp.p TRUE
```

```
SECOND := λp.p FALSE
```

```
NIL := λx.TRUE
```

```
NULL := λp.p (λx.λy.FALSE)
```

A linked list can be defined as either NIL for the empty list, or the PAIR of an element and a smaller list. The predicate NULL tests for the value NIL. (Alternatively, with NIL := FALSE, the construct $\perp (\lambda h. \lambda t. \lambda z. \text{deal_with_head_h_and_tail_t}) (\text{deal_with_nil})$ obviates the need for an explicit NULL test).

As an example of the use of pairs, the shift-and-increment function that maps (m, n) to $(n, n + 1)$ can be defined as

```
Φ := λx.PAIR (SECOND x) (SUCC (SECOND x))
```

which allows us to give perhaps the most transparent version of the predecessor function:

```
PRED := λn.FIRST (n Φ (PAIR 0 0)).
```

Recursion and fixed points

Recursion is the definition of a function using the function itself; on the face of it, lambda calculus does not allow this (we can't refer to a value which is yet to be defined, inside the lambda term defining that same value, as all functions are anonymous in lambda calculus). However, this impression is misleading: in $(\lambda x. x x) y$ both x 's refer to the same lambda term, y , so it is possible for a lambda expression – here y – to be arranged to receive itself as its argument value, through self-application.

Consider for instance the factorial function $F(n)$ recursively defined by

```
F(n) = 1, if n = 0; else n × F(n - 1).
```

In the lambda expression which is to represent this function, a *parameter* (typically the first one) will be assumed to receive the lambda expression itself as its value, so that calling it – applying it to an argument – will amount to recursion. Thus to achieve recursion, the intended-as-self-referencing argument (called r here) must always be passed to itself within the function body, at a call point:

```
G := λr. λn.(1, if n = 0; else n × (r r (n-1)))
```

with $r r x = F x = G r x$ to hold, so $r = G$ and

```
F := G G = (λx.x x) G
```

The self-application achieves replication here, passing the function's lambda expression on to the next invocation as an argument value, making it available to be referenced and called there.

This solves the specific problem of the factorial function, but we'd like to have a generic solution, i.e. not forcing a specific re-write for every recursive function:

```
G := λr. λn.(1, if n = 0; else n × (r (n-1)))
with r x = F x = G r x to hold, so r = G r =: FIX G and
F := FIX G where FIX g := (r where r = g r) = g (FIX g)
so that FIX G = G (FIX G) = (λn.(1, if n = 0; else n × ((FIX G)
(n-1))))
```

Given a lambda term with first argument representing recursive call (e.g. G here), the *fixed-point* combinator FIX will return a self-replicating lambda expression representing the recursive function (here, F). The function does not need to be explicitly passed to itself at any point, for the self-replication is arranged in advance, when it is created, to be done each time it is called. Thus the original lambda expression ($\text{FIX } G$) is re-created inside itself, at call-point, achieving self-reference.

In fact, there are many possible definitions for this FIX operator, the simplest of them being:

```
Y := λg. (λx.g (x x)) (λx.g (x x))
```

In the lambda calculus, $\mathbf{Y} g$ is a fixed-point of g , as it expands to:

```
Y g
λh. ((λx.h (x x)) (λx.h (x x))) g
(λx.g (x x)) (λx.g (x x))
g ((λx.g (x x)) (λx.g (x x)))
g (Y g)
```

Now, to perform our recursive call to the factorial function, we would simply call $(\mathbf{Y} \ G) \ n$, where n is the number we are calculating the factorial of. Given $n = 4$, for example, this gives:

```
(Y G) 4
G (Y G) 4
(λr.λn.(1, if n = 0; else n × (r (n-1)))) (Y G) 4
(λn.(1, if n = 0; else n × ((Y G) (n-1)))) 4
1, if 4 = 0; else 4 × ((Y G) (4-1))
4 × (G (Y G) (4-1))
4 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (4-1))
4 × (1, if 3 = 0; else 3 × ((Y G) (3-1)))
4 × (3 × (G (Y G) (3-1)))
4 × (3 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (3-1)))
4 × (3 × (1, if 2 = 0; else 2 × ((Y G) (2-1))))
4 × (3 × (2 × (G (Y G) (2-1))))
4 × (3 × (2 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (2-1))))
4 × (3 × (2 × (1, if 1 = 0; else 1 × ((Y G) (1-1)))))
4 × (3 × (2 × (1 × (G (Y G) (1-1)))))
4 × (3 × (2 × (1 × ((λn.(1, if n = 0; else n × ((Y G) (n-1)))) (1-1)))))
4 × (3 × (2 × (1 × (1, if 0 = 0; else 0 × ((Y G) (0-1))))))
4 × (3 × (2 × (1 × (1))))
```

Every recursively defined function can be seen as a fixed point of some suitably defined function closing over the recursive call with an extra argument, and therefore, using \mathbf{Y} , every recursively defined function can be expressed as a lambda expression. In particular, we can now cleanly define the subtraction, multiplication and comparison predicate of natural numbers recursively.

Standard terms

Certain terms have commonly accepted names:

$$\begin{aligned}\mathbf{I} &:= \lambda x. x \\ \mathbf{K} &:= \lambda x. \lambda y. x \\ \mathbf{S} &:= \lambda x. \lambda y. \lambda z. x z (y z) \\ \mathbf{B} &:= \lambda x. \lambda y. \lambda z. x (y z) \\ \mathbf{C} &:= \lambda x. \lambda y. \lambda z. x z y \\ \mathbf{W} &:= \lambda x. \lambda y. x y y \\ \mathbf{\omega} &:= \lambda x. x x \\ \mathbf{\Omega} &:= \mathbf{\omega} \mathbf{\omega} \\ \mathbf{Y} &:= \lambda g. (\lambda x. g (x x)) (\lambda x. g (x x))\end{aligned}$$

Computable functions and lambda calculus

A function $F: \mathbf{N} \rightarrow \mathbf{N}$ of natural numbers is a computable function if and only if there exists a lambda expression f such that for every pair of x, y in \mathbf{N} , $F(x)=y$ if and only if $f x =_{\beta} y$, where x and y are the Church numerals corresponding to x and y , respectively and $=_{\beta}$ meaning equivalence with beta reduction. This is one of the many ways to define computability; see the Church-Turing thesis for a discussion of other approaches and their equivalence.

Undecidability of equivalence

There is no algorithm that takes as input two lambda expressions and outputs TRUE or FALSE depending on whether or not the two expressions are equivalent. This was historically the first problem for which undecidability could be proven. As is common for a proof of undecidability, the proof shows that no computable function can decide the equivalence. Church's thesis is then invoked to show that no algorithm can do so.

Church's proof first reduces the problem to determining whether a given lambda expression has a *normal form*. A normal form is an equivalent expression that cannot be reduced any further under the rules imposed by the form. Then he assumes that this predicate is computable, and can hence be expressed in lambda calculus. Building on earlier work by Kleene and constructing a Gödel numbering for lambda expressions, he constructs a lambda expression e that closely follows the proof of Gödel's first incompleteness theorem. If e is applied to its own Gödel number, a contradiction results.

Lambda calculus and programming languages

As pointed out by Peter Landin's 1965 paper A Correspondence between ALGOL 60 and Church's Lambda-notation [14], sequential procedural programming languages can be understood in terms of the lambda calculus, which provides the basic mechanisms for procedural abstraction and procedure (subprogram) application.

Lambda calculus reifies "functions" and makes them first-class objects, which raises implementation complexity when it is implemented.

Anonymous functions

For example in Lisp the 'square' function can be expressed as a lambda expression as follows:

```
(lambda (x) (* x x))
```

or the same expressed in Haskell:

```
\x -> x*x -- where the \ denotes the greek λ
```

The above example is an expression that evaluates to a first-class function. The symbol `lambda` creates an anonymous function, given a list of parameter names, `(x)` — just a single argument in this case, and an expression that is evaluated as the body of the function, `(* x x)`. The Haskell example is identical. Anonymous functions are sometimes called lambda expressions.

For example Pascal and many other imperative languages have long supported passing subprograms as arguments to other subprograms through the mechanism of function pointers. However, function pointers are not sufficient condition for functions to be first class datatype because if and only if new instances of a function can be created at run time, the functions are first class datatype. And this run-time creation of functions is supported in C++, Smalltalk, and more recently in Scala, Eiffel ("agents") and C# ("delegates"), among others.

Below is an example expressed as the Eiffel "inline agent"

```
agent (x: REAL) : REAL do Result := x * x end
```

The object corresponds to the lambda expression $\lambda x.x^*x$ (with call by value) because it can be assigned to a variable or passed around to routines, i.e. treated like any other expression.

A Python example of this uses the `lambda` [15] form of functions:

```
func = lambda x: x * x
```

This creates a new anonymous function and names it `func` that can be passed to other functions, stored in variables, etc. Python can also treat any other function created with the standard `def` [16] statement as first-class objects.

The same holds for Smalltalk expression

```
[ :x | x * x ]
```

This is first-class object (block closure), which can be stored in variables, passed as arguments, etc.

A similar expression using C++11 anonymous function, but specifically for integers, is:

```
[] (int i) -> int { return i * i; }
```

In JavaScript since version 1.8, the notation:

```
function(x) x * x;
```

is used. In older versions

```
function(x) { return x * x; }
```

In Scala:

```
(x:Int) => x * x
```

In OCaml (and F#):

```
fun x -> x * x
```

In D:

```
x => x * x           // when parameter type can be inferred
(int x) => x * x    // when parameter type must be specified
```

In Ruby:

```
lambda { |x| x * x }    #Ruby prior to 1.9
-> (x) { x * x }        #Ruby 1.9
```

In C#:

```
Func<int, int> Square = x => x * x;
```

Reduction strategies

Whether a term is normalising or not, and how much work needs to be done in normalising it if it is, depends to a large extent on the reduction strategy used. The distinction between reduction strategies relates to the distinction in functional programming languages between eager evaluation and lazy evaluation.

Full beta reductions

Any redex can be reduced at any time. This means essentially the lack of any particular reduction strategy—with regard to reducibility, "all bets are off".

Applicative order

The rightmost, innermost redex is always reduced first. Intuitively this means a function's arguments are always reduced before the function itself. Applicative order always attempts to apply functions to normal forms, even when this is not possible.

Most programming languages (including Lisp, ML and imperative languages like C and Java) are described as "strict", meaning that functions applied to non-normalising arguments are non-normalising. This is done essentially using applicative order, call by value reduction (see below), but usually called "eager evaluation".

Normal order

The leftmost, outermost redex is always reduced first. That is, whenever possible the arguments are substituted into the body of an abstraction before the arguments are reduced.

Call by name

As normal order, but no reductions are performed inside abstractions. For example $\lambda x. (\lambda x. x) x$ is in normal form according to this strategy, although it contains the redex $(\lambda x. x) x$.

Call by value

Only the outermost redexes are reduced: a redex is reduced only when its right hand side has reduced to a value (variable or lambda abstraction).

Call by need

As normal order, but function applications that would duplicate terms instead name the argument, which is then reduced only "when it is needed". Called in practical contexts "lazy evaluation". In implementations this "name" takes the form of a pointer, with the redex represented by a thunk.

Applicative order is not a normalising strategy. The usual counterexample is as follows: define $\Omega = \omega\omega$ where $\omega = \lambda x. xx$. This entire expression contains only one redex, namely the whole expression; its reduct is again Ω . Since this is the only available reduction, Ω has no normal form (under any evaluation strategy). Using applicative order, the expression $KI\Omega = (\lambda x. \lambda y. x) (\lambda x. x)\Omega$ is reduced by first reducing Ω to normal form (since it is the rightmost redex), but since Ω has no normal form, applicative order fails to find a normal form for $KI\Omega$.

In contrast, normal order is so called because it always finds a normalising reduction, if one exists. In the above example, $KI\Omega$ reduces under normal order to I , a normal form. A drawback is that redexes in the arguments may be copied, resulting in duplicated computation (for example, $(\lambda x. xx) ((\lambda x. x)y)$ reduces to $((\lambda x. x)y) ((\lambda x. x)y)$ using this strategy; now there are two redexes, so full evaluation needs two more steps, but if the argument had been reduced first, there would now be none).

The positive tradeoff of using applicative order is that it does not cause unnecessary computation, if all arguments are used, because it never substitutes arguments containing redexes and hence never needs to copy them (which would duplicate work). In the above example, in applicative order $(\lambda x. xx) ((\lambda x. x)y)$ reduces first to $(\lambda x. xx)y$ and then to the normal order yy , taking two steps instead of three.

Most *purely* functional programming languages (notably Miranda and its descendants, including Haskell), and the proof languages of theorem provers, use *lazy evaluation*, which is essentially the same as call by need. This is like normal order reduction, but call by need manages to avoid the duplication of work inherent in normal order reduction using *sharing*. In the example given above, $(\lambda x. xx) ((\lambda x. x)y)$ reduces to $((\lambda x. x)y) ((\lambda x. x)y)$, which has two redexes, but in call by need they are represented using the same object rather than copied, so when one is reduced the other is too.

A note about complexity

While the idea of beta reduction seems simple enough, it is not an atomic step, in that it must have a non-trivial cost when estimating computational complexity.^[17] To be precise, one must somehow find the location of all of the occurrences of the bound variable V in the expression E , implying a time cost, or one must keep track of these locations in some way, implying a space cost. A naïve search for the locations of V in E is $O(n)$ in the length n of E . This has led to the study of systems that use explicit substitution. Sinot's director strings^[18] offer a way of tracking the locations of free variables in expressions.

Parallelism and concurrency

The Church-Rosser property of the lambda calculus means that evaluation (β -reduction) can be carried out in *any order*, even in parallel. This means that various nondeterministic evaluation strategies are relevant. However, the lambda calculus does not offer any explicit constructs for parallelism. One can add constructs such as Futures to the lambda calculus. Other process calculi have been developed for describing communication and concurrency.

Semantics

The fact that lambda calculus terms act as functions on other lambda calculus terms, and even on themselves, led to questions about the semantics of the lambda calculus. Could a sensible meaning be assigned to lambda calculus terms? The natural semantics was to find a set D isomorphic to the function space $D \rightarrow D$, of functions on itself. However, no nontrivial such D can exist, by cardinality constraints because the set of all functions from D into D has greater cardinality than D .

In the 1970s, Dana Scott showed that, if only continuous functions were considered, a set or domain D with the required property could be found, thus providing a model for the lambda calculus.

This work also formed the basis for the denotational semantics of programming languages.

References

- [1] A. Church, "A set of postulates for the foundation of logic", *Annals of Mathematics*, Series 2, 33:346–366 (1932).
- [2] For a full history, see Cardone and Hindley's "History of Lambda-calculus and Combinatory Logic" (2006).
- [3] A. Church, "An unsolvable problem of elementary number theory", *American Journal of Mathematics*, Volume 58, No. 2. (April 1936), pp. 345–363.
- [4] A. Church, "A Formulation of the Simple Theory of Types", *Journal of Symbolic Logic*, Volume 5 (1940).
- [5] Barendregt, Hendrik Pieter (1984), *The Lambda Calculus: Its Syntax and Semantics* (http://www.elsevier.com/wps/find/bookdescription.cws_home/501727/description), Studies in Logic and the Foundations of Mathematics, **103** (Revised ed.), North Holland, Amsterdam. Corrections (<ftp://ftp.cs.ru.nl/pub/CompMath.Found/ErrataLCalculus.pdf>), ISBN 0-444-87508-5,
- [6] "Example for Rules of Associativity" (<http://www.lambda-bound.com/book/lambdacalc/node27.html>). Lambda-bound.com. . Retrieved 2012-06-18.
- [7] Selinger, Peter, *Lecture Notes on the Lambda Calculus* (<http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf>), Department of Mathematics and Statistics, University of Ottawa, p. 9,
- [8] "Example for Rule of Associativity" (<http://www.lambda-bound.com/book/lambdacalc/node25.html>). Lambda-bound.com. . Retrieved 2012-06-18.
- [9] Barendregt, Henk; Barendsen, Erik (March 2000), *Introduction to Lambda Calculus* (<ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf>),
- [10] de Queiroz, Ruy J.G.B. " A Proof-Theoretic Account of Programming and the Role of Reduction Rules. (<http://dx.doi.org/10.1111/j.1746-8361.1988.tb00919.x>)" *Dialectica* **42**(4), pages 265-282, 1988.
- [11] Turbak, Franklyn; Gifford, David (2008), *Design concepts in programming languages*, MIT press, p. 251, ISBN 978-0-262-20175-9
- [12] Felleisen, Matthias; Flatt, Matthew (2006), *Programming Languages and Lambda Calculi* (<http://www.cs.utah.edu/plt/publications/pllc.pdf>), p. 26,
- [13] Selinger, Peter, *Lecture Notes on the Lambda Calculus* (<http://www.mathstat.dal.ca/~selinger/papers/lambdanotes.pdf>), Department of Mathematics and Statistics, University of Ottawa, p. 16,
- [14] <http://portal.acm.org/citation.cfm?id=363749&coll=portal&dl=ACM>
- [15] <http://docs.python.org/ref/lambdas.html#lambda>
- [16] <http://docs.python.org/ref/function.html>
- [17] R. Statman, " The typed λ -calculus is not elementary recursive. (http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4567929)" *Theoretical Computer Science*, (1979) **9** pp73-81.
- [18] F.-R. Sinot. " Director Strings Revisited: A Generic Approach to the Efficient Representation of Free Variables in Higher-order Rewriting. (<http://www.lsv.ens-cachan.fr/~sinot/publis.php?onlykey=sinot-jlc05>)" *Journal of Logic and Computation* **15**(2), pages 201-218, 2005.

Further reading

- Abelson, Harold & Gerald Jay Sussman. Structure and Interpretation of Computer Programs. The MIT Press. ISBN 0-262-51087-1.
- Hendrik Pieter Barendregt *Introduction to Lambda Calculus* (<ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambda.pdf>).
- Henk Barendregt, The Impact of the Lambda Calculus in Logic and Computer Science (http://people.emich.edu/pstephen/other_papers/Impact%20of%20the%20Lambda%20Calculus.pdf). The Bulletin of Symbolic Logic, Volume 3, Number 2, June 1997.
- Barendregt, Hendrik Pieter, *The Type Free Lambda Calculus* pp1091–1132 of *Handbook of Mathematical Logic*, North-Holland (1977) ISBN 0-7204-2285-X
- Cardone and Hindley, 2006. History of Lambda-calculus and Combinatory Logic (<http://www-maths.swan.ac.uk/staff/jrh/papers/JRHIslamWeb.pdf>). In Gabbay and Woods (eds.), *Handbook of the History of Logic*, vol. 5. Elsevier.
- Church, Alonzo, *An unsolvable problem of elementary number theory*, American Journal of Mathematics, 58 (1936), pp. 345–363. This paper contains the proof that the equivalence of lambda expressions is in general not decidable.
- Kleene, Stephen, *A theory of positive integers in formal logic*, American Journal of Mathematics, 57 (1935), pp. 153–173 and 219–244. Contains the lambda calculus definitions of several familiar functions.
- Landin, Peter, *A Correspondence Between ALGOL 60 and Church's Lambda-Notation*, Communications of the ACM, vol. 8, no. 2 (1965), pages 89–101. Available from the ACM site (<http://portal.acm.org/citation>).

cfm?id=363749&coll=portal&dl=ACM). A classic paper highlighting the importance of lambda calculus as a basis for programming languages.

- Larson, Jim, *An Introduction to Lambda Calculus and Scheme* (<http://www.jetcafe.org/~jim/lambda.html>). A gentle introduction for programmers.
- Schalk, A. and Simmons, H. (2005) *An introduction to λ -calculi and arithmetic with a decent selection of exercises* (<http://www.cs.man.ac.uk/~hsimmons/BOOKS/lcalculus.pdf>). *Notes for a course in the Mathematical Logic MSc at Manchester University*.
- de Queiroz, Ruy J.G.B. (2008) *On Reduction Rules, Meaning-as-Use and Proof-Theoretic Semantics* (<http://www.springerlink.com/content/27nk266126k817gq/>). *Studia Logica*, 90(2):211-247. A paper giving a formal underpinning to the idea of ‘meaning-is-use’ which, even if based on proofs, it is different from proof-theoretic semantics as in the Dummett–Prawitz tradition since it takes reduction as the rules giving meaning.

Monographs/textbooks for graduate students:

- Morten Heine Sørensen, Paweł Urzyczyn, *Lectures on the Curry-Howard isomorphism*, Elsevier, 2006, ISBN 0-444-52077-5 is a recent monograph that covers the main topics of lambda calculus from the type-free variety, to most typed lambda calculi, including more recent developments like pure type systems and the lambda cube. It does not cover subtyping extensions.
- Pierce, Benjamin (2002), *Types and Programming Languages*, MIT Press, ISBN 0-262-16209-1 covers lambda calculi from a practical type system perspective; some topics like dependent types are only mentioned, but subtyping is an important topic.

Some parts of this article are based on material from FOLDOC, used with permission.

External links

- Achim Jung, *A Short Introduction to the Lambda Calculus* (<http://www.cs.bham.ac.uk/~axj/pub/papers/lambda-calculus.pdf>)-(PDF)
- David C. Keenan, *To Dissect a Mockingbird: A Graphical Notation for the Lambda Calculus with Animated Reduction* (<http://dkeenan.com/Lambda/>)
- Raúl Rojas, *A Tutorial Introduction to the Lambda Calculus* (<http://www.inf.fu-berlin.de/lehre/WS03/alpi/lambda.pdf>)-(PDF)
- Peter Selinger, *Lecture Notes on the Lambda Calculus* (<http://www.mscs.dal.ca/~selinger/papers/#lambdanotes>)-(PDF)
- L. Allison, *Some executable λ -calculus examples* (<http://www.allisons.org/ll/FP/Lambda/Examples/>)
- Georg P. Loczewski, *The Lambda Calculus and A++* (<http://www.lambda-bound.com/book/lambdacalc/lcalconl.html>)
- Bret Victor, *Alligator Eggs: A Puzzle Game Based on Lambda Calculus* (<http://worrydream.com/AlligatorEggs/>)
- *Lambda Calculus* (<http://www.safalra.com/science/lambda-calculus/>) on Safalra’s Website (<http://www.safalra.com/>)
- *Lambda Calculus* (<http://planetmath.org/?op=getobj&from=objects&id=2788>), *PlanetMath.org*.
- LCI Lambda Interpreter (<http://lci.sourceforge.net/>) a simple yet powerful pure calculus interpreter
- Lambda Calculus links on Lambda-the-Ultimate (<http://lambda-the-ultimate.org/classic/lc.html>)
- Mike Thyer, Lambda Animator (<http://thyer.name/lambda-animator/>), a graphical Java applet demonstrating alternative reduction strategies.
- An Introduction to Lambda Calculus and Scheme (<http://www.jetcafe.org/~jim/lambda.html>), by Jim Larson
- Implementing the Lambda calculus (<http://matt.might.net/articles/c++-template-meta-programming-with-lambda-calculus/>) using C++ Templates

Closure (computer science)

In computer science, a **closure** (also **lexical closure** or **function closure**) is a function or reference to a function together with a *referencing environment*—a table storing a reference to each of the non-local variables (also called free variables) of that function.^[1] A closure—unlike a plain function pointer—allows a function to access those non-local variables even when invoked outside of its immediate lexical scope.

The concept of closures was developed in the 1960s and was first fully implemented in 1975 as a language feature in the Scheme programming language to support lexically scoped first-class functions. The explicit use of closures is associated with functional programming languages such as Lisp and ML, as traditional imperative languages such as Algol, C and Pascal did not support returning nested functions as results of higher-order functions and thus did not require supporting closures either. Many modern garbage-collected imperative languages (such as Smalltalk, the first object-oriented language featuring closures,^[2] C#, but notably not Java^[3]) support closures.

Example

The following fragment of Python 3 code defines a function `counter` with a local variable `x` and a nested function `increment`. This nested function `increment` has access to `x`, which from its point of view is a non-local variable. The function `counter` returns a closure containing a reference to the function `increment` and `increment`'s non-local variable `x`.

```
def counter():
    x = 0
    def increment(y):
        nonlocal x
        x += y
        print(x)
    return increment
```

The closure returned by `counter` can be assigned to a variable:

```
counter1_increment = counter()
counter2_increment = counter()
```

Invoking `increment` through the closures will give the following results:

```
counter1_increment(1)      # prints 1
counter1_increment(7)      # prints 8
counter2_increment(1)      # prints 1
counter1_increment(1)      # prints 9
```

History and etymology

Peter J. Landin defined the term *closure* in 1964 as having an *environment part* and a *control part* as used by his SECD machine for evaluating expressions.^[4] Joel Moses credits Landin with introducing the term *closure* to refer to a lambda expression whose open bindings (free variables) have been closed by (or bound in) the lexical environment, resulting in a *closed expression*, or closure.^{[5][6]} This usage was subsequently adopted by Sussman and Steele when they defined Scheme in 1975,^[7] and became widespread.

The term *closure* is often mistakenly used to mean anonymous function. This is probably because most languages implementing anonymous functions allow them to form closures and programmers are usually introduced to both

concepts at the same time. An anonymous function can be seen as a function *literal*, while a closure is a function *value*. These are, however, distinct concepts. A closure retains a reference to the environment at the time it was created (for example, to the current value of a local variable in the enclosing scope) while a generic anonymous function need not do this.

Implementation and theory

Closures are typically implemented with a special data structure that contains a pointer to the function code, plus a representation of the function's lexical environment (e.g., the set of available variables and their values) at the time when the closure was created. The referencing environment binds the nonlocal names to the corresponding variables in scope at the time the closure is created, additionally extending their lifetime to at least as long as the lifetime of the closure itself. When the closure is *entered* at a later time, possibly from a different scope, the function is executed with its non-local variables referring to the ones captured by the closure.

A language implementation cannot easily support full closures if its run-time memory model allocates all local variables on a linear stack. In such languages, a function's local variables are deallocated when the function returns. However, a closure requires that the free variables it references survive the enclosing function's execution. Therefore, those variables must be allocated so that they persist until no longer needed. This explains why, typically, languages that natively support closures also use garbage collection. The alternative is for the language to accept that certain use cases will lead to undefined behaviour, as in the proposal for lambda expressions in C++.^[8] The Funarg problem (or "functional argument" problem) describes the difficulty of implementing functions as first class objects in a stack-based programming language such as C or C++. Similarly in D version 1, it is assumed that the programmer knows what to do with delegates and local variables, as their references will be invalid after return from its definition scope (local variables are on the stack) - this still permits many useful functional patterns, but for complex cases needs explicit heap allocation for variables. D version 2 solved this by detecting which variables must be stored on the heap, and performs automatic allocation. Because D uses garbage collection, in both versions, there is no need to track usage of variables as they are passed.

In strict functional languages with immutable data (e.g. Erlang), it is very easy to implement automatic memory management (garbage collection), as there are no possible cycles in variables references. For example in Erlang, all arguments and variables are allocated on the heap, but references to them are additionally stored on the stack. After a function returns, references are still valid. Heap cleaning is done by incremental garbage collector.

In ML, local variables are allocated on a linear stack . When a closure is created, it copies the values of those variables that are needed by the closure into the closure's data structure.

Scheme, which has an ALGOL-like lexical scope system with dynamic variables and garbage collection, lacks a stack programming model and does not suffer from the limitations of stack-based languages. Closures are expressed naturally in Scheme. The lambda form encloses the code and the free variables of its environment, persists within the program as long as it can possibly be accessed, and can be used as freely as any other Scheme expression.

Closures are closely related to Actors in the Actor model of concurrent computation where the values in the function's lexical environment are called *acquaintances*. An important issue for closures in concurrent programming languages is whether the variables in a closure can be updated and, if so, how these updates can be synchronized. Actors provide one solution.^[9]

Closures are closely related to function objects; the transformation from the former to the latter is known as defunctionalization or lambda lifting.

Applications

Closures are used to implement continuation-passing style, and in this manner, hide state. Constructs such as objects and control structures can thus be implemented with closures. In some languages, a closure may occur when a function is defined within another function, and the inner function refers to local variables of the outer function. At run-time, when the outer function executes, a closure is formed, consisting of the inner function's code and references (the upvalues) to any variables of the outer function required by the closure.

First-class functions

Further information: First-class function

Closures typically appear in languages in which functions are first-class values—in other words, such languages allow functions to be passed as arguments, returned from function calls, bound to variable names, etc., just like simpler types such as strings and integers. For example, consider the following Scheme function:

```
; Return a list of all books with at least THRESHOLD copies sold.
(define (best-selling-books threshold)
  (filter
    (lambda (book)
      (>= (book-sales book) threshold))
    book-list))
```

In this example, the lambda expression `(lambda (book) (>= (book-sales book) threshold))` appears within the function `best-selling-books`. When the lambda expression is evaluated, Scheme creates a closure consisting of the code for the lambda expression and a reference to the `threshold` variable, which is a free variable inside the lambda expression.

The closure is then passed to the `filter` function, which calls it repeatedly to determine which books are to be added to the result list and which are to be discarded. Because the closure itself has a reference to `threshold`, it can use that variable each time `filter` calls it. The function `filter` itself might be defined in a completely separate file.

Here is the same example rewritten in JavaScript, another popular language with support for closures:

```
// Return a list of all books with at least 'threshold' copies sold.
function bestSellingBooks(threshold) {
  return bookList.filter(
    function (book) { return book.sales >= threshold; }
  );
}
```

The `function` keyword is used here instead of `lambda`, and an `Array.filter` method^[10] instead of a global `filter` function, but otherwise the structure and the effect of the code are the same.

A function may create a closure and return it, as in the following example:

```
// Return a function that approximates the derivative of f
// using an interval of dx, which should be appropriately small.
function derivative(f, dx) {
  return function (x) {
    return (f(x + dx) - f(x)) / dx;
  };
}
```

Because the closure in this case outlives the scope of the function that creates it, the variables `f` and `dx` live on after the function `derivative` returns. In languages without closures, the lifetime of a local variable coincides with the execution of the scope where that variable is declared. In languages with closures, variables must continue to exist as long as any existing closures have references to them. This is most commonly implemented using some form of garbage collection.

State representation

A closure can be used to associate a function with a set of "private" variables, which persist over several invocations of the function. The scope of the variable encompasses only the closed-over function, so it cannot be accessed from other program code.

In stateful languages, closures can thus be used to implement paradigms for state representation and information hiding, since the closure's upvalues (its closed-over variables) are of indefinite extent, so a value established in one invocation remains available in the next. Closures used in this way no longer have referential transparency, and are thus no longer pure functions; nevertheless, they are commonly used in "near-functional" languages such as Scheme.

Other uses

Closures have many uses:

- Because closures delay evaluation—i.e., they do not "do" anything until they are called—they can be used to define control structures. For example, all Smalltalk's standard control structures, including branches (`if/then/else`) and loops (`while` and `for`), are defined using objects whose methods accept closures. Users can easily define their own control structures also.
- In languages that allow assignment, multiple functions can be produced that close over the same environment, enabling them to communicate privately by altering that environment. In Scheme:

```
(define foo #f)
(define bar #f)

(let ((secret-message "none"))
  (set! foo (lambda (msg) (set! secret-message msg)))
  (set! bar (lambda () secret-message)))

(display (bar)) ; prints "none"
(newline)
(foo "meet me by the docks at midnight")
(display (bar)) ; prints "meet me by the docks at midnight"
```

- Closures can be used to implement object systems.^[11]

Note: Some speakers call any data structure that binds a lexical environment a closure, but the term usually refers specifically to functions.

Differences in semantics

Lexical environment

As different languages do not always have a common definition of the lexical environment, their definitions of closure may vary also. The commonly held minimalist definition of the lexical environment defines it as a set of all bindings of variables in the scope, and that is also what closures in any language have to capture. However the meaning of a variable binding also differs. In imperative languages, variables bind to relative locations in memory that can store values. Although the relative location of a binding does not change at runtime, the value in the bound location can. In such languages, since closure captures the binding, any operation on the variable, whether done from the closure or not, are performed on the same relative memory location. This is often called capturing the variable "by reference". Here is an example illustrating the concept in ECMAScript, which is one such language:

```
// ECMAScript
var f, g;
function foo() {
    var x = 0;
    f = function() { return ++x; };
    g = function() { return --x; };
    x = 1;
    alert('inside foo, call to f(): ' + f()); // "2"
}
foo();
alert('call to g(): ' + g()); // "1"
alert('call to f(): ' + f()); // "2"
```

Note how function `foo` and the closures referred to by variables `f` and `g` all use the same relative memory location signified by local variable `x`.

On the other hand, many functional languages, such as ML, bind variables directly to values. In this case, since there is no way to change the value of the variable once it is bound, there is no need to share the state between closures—they just use the same values. This is often called capturing the variable "by value". Java's local and anonymous classes also fall into this category -- they required captured local variables to be `final`, which also means there is no need to share state.

Some languages allow you to choose between capturing the value of a variable or its location. For example, in C++11 and PHP, captured variables are either declared with `&`, which means captured by reference, or without, which means captured by value.

Yet another subset, lazy functional languages such as Haskell, bind variables to results of future computations rather than values. Consider this example in Haskell:

```
-- Haskell
foo :: Num -> Num -> (Num -> Num)
foo x y = let r = x / y
          in (\z -> z + r)

f :: Num -> Num
f = foo 1 0

main = print (f 123)
```

The binding of `x` captured by the closure defined within function `foo` is to the computation `(x / y)` - which in this case results in division by zero. However, since it is the computation that is captured, and not the value, the error only manifests itself when the closure is invoked, and actually attempts to use the captured binding.

Closure leaving

Yet more differences manifest themselves in the behavior of other lexically scoped constructs, such as `return`, `break` and `continue` statements. Such constructs can, in general, be considered in terms of invoking an escape continuation established by an enclosing control statement (in case of `break` and `continue`, such interpretation requires looping constructs to be considered in terms of recursive function calls). In some languages, such as ECMAScript, `return` refers to the continuation established by the closure lexically innermost with respect to the statement—thus, a `return` within a closure transfers control to the code that called it. However in Smalltalk, the superficially similar operator `^` invokes the escape continuation established for the method invocation, ignoring the escape continuations of any intervening nested closures. The escape continuation of a particular closure can only be invoked in Smalltalk implicitly by reaching the end of the closure's code. The following examples in ECMAScript and Smalltalk highlight the difference:

```
"Smalltalk"
foo
| xs |
xs := #(1 2 3 4).
xs do: [:x | ^x].
^0
bar
Transcript show: (self foo printString) "prints 1"

// ECMAScript
function foo() {
  var xs = [1, 2, 3, 4];
  xs.forEach(function (x) { return x; });
  return 0;
}
alert(foo()); // prints 0
```

The above code snippets will behave differently because the Smalltalk `^` operator and the JavaScript `return` operator are not analogous. In the ECMAScript example, `return x` will leave the inner closure to begin a new iteration of the `forEach` loop, whereas in the Smalltalk example, `^x` will abort the loop and return from the method `foo`.

Common Lisp provides a construct that can express either of the above actions: Lisp `(return-from foo x)` behaves as Smalltalk `^x`, while Lisp `(return-from nil x)` behaves as JavaScript `return x`. Hence, Smalltalk makes it possible for a captured escape continuation to outlive the extent in which it can be successfully invoked. Consider:

```
"Smalltalk"
foo
  ^[ :x | ^x ]
bar
| f |
f := self foo.
f value: 123 "error!"
```

When the closure returned by the method `foo` is invoked, it attempts to return a value from the invocation of `foo` that created the closure. Since that call has already returned and the Smalltalk method invocation model does not follow the spaghetti stack discipline to allow multiple returns, this operation results in an error.

Some languages, such as Ruby, allow the programmer to choose the way `return` is captured. An example in Ruby:

```
# Ruby

# Closure using a Proc
def foo
  f = Proc.new { return "return from foo from inside proc" }
  f.call # control leaves foo here
  return "return from foo"
end

# Closure using a lambda
def bar
  f = lambda { return "return from lambda" }
  f.call # control does not leave bar here
  return "return from bar"
end

puts foo # prints "return from foo from inside proc"
puts bar # prints "return from bar"
```

Both `Proc.new` and `lambda` in this example are ways to create a closure, but semantics of the closures thus created are different with respect to the `return` statement.

In Scheme, definition and scope of the `return` control statement is explicit (and only arbitrarily named 'return' for the sake of the example). The following is a direct translation of the Ruby sample.

```
; Scheme

(define call/cc call-with-current-continuation)

(define (foo)
  (call/cc
    (lambda (return)
      (define (f) (return "return from foo from inside proc"))
      (f) ; control leaves foo here
      (return "return from foo"))))

(define (bar)
  (call/cc
    (lambda (return)
      (define (f) (call/cc (lambda (return) (return "return from
lambda")))))
      (f) ; control does not leave bar here
      (return "return from bar"))))
```

```
(display (foo)) ; prints "return from foo from inside proc"
(newline)
(display (bar)) ; prints "return from bar"
```

Closure-like constructs

Features of some languages simulate some features of closures. Language features include some object-oriented techniques, for example in Java, C++, Objective-C, C#, D.

Callbacks (C)

In C, libraries that support callbacks sometimes allow a callback to be registered using two values: a function pointer and a separate `void*` pointer to arbitrary data of the user's choice. Each time the library executes the callback function, it passes in the data pointer. This allows the callback to maintain state and to refer to information captured at the time it was registered. The idiom is similar to closures in functionality, but not in syntax.

Local classes (Java)

Java allows classes to be defined inside methods. These are called *local classes*. When such classes are not named, they are known as *anonymous classes* (or anonymous *inner* classes). A local class (either named or anonymous) may refer to names in lexically enclosing classes, or read-only variables (marked as `final`) in the lexically enclosing method.

```
class CalculationWindow extends JFrame {
    private volatile int result;
    ...
    public void calculateInSeparateThread(final URI uri) {
        // The expression "new Runnable() { ... }" is an anonymous class
        // implementing the 'Runnable' interface.
        new Thread(
            new Runnable() {
                void run() {
                    // It can read final local variables:
                    calculate(uri);
                    // It can access private fields of the enclosing class:
                    result = result + 10;
                }
            }
        ).start();
    }
}
```

The capturing of `final` variables allows you to capture variables by value. Even if the variable you want to capture is non-`final`, you can always copy it to a temporary `final` variable just before the class.

Capturing of variables by reference can be emulated by using a `final` reference to a mutable container, for example, a single-element array. The local class will not be able to change the value of the container reference itself, but it will be able to change the contents of the container.

According to a Java 8 proposal,^[12] closures will allow the above code to be executed as:

```
class CalculationWindow extends JFrame {  
    private volatile int result;  
    ...  
    public void calculateInSeparateThread(final URI uri) {  
        // the code () -> { /* code */ } is a closure  
        new Thread(() -> {  
            calculate(uri);  
            result = result + 10;  
        }).start();  
    }  
}
```

Local classes are one of the types of inner class that are declared within the body of a method. Java also supports inner classes that are declared as *non-static members* of an enclosing class.^[13] They are normally referred to just as "inner classes".^[14] These are defined in the body of the enclosing class and have full access to instance variables of the enclosing class. Due to their binding to these instance variables, an inner class may only be instantiated with an explicit binding to an instance of the enclosing class using a special syntax.^[15]

```
public class EnclosingClass {  
    /* Define the inner class */  
    public class InnerClass {  
        public int incrementAndReturnCounter() {  
            return counter++;  
        }  
    }  
  
    private int counter;  
  
    {  
        counter = 0;  
    }  
  
    public int getCounter() {  
        return counter;  
    }  
  
    public static void main(String[] args) {  
        EnclosingClass enclosingClassInstance = new EnclosingClass();  
        /* Instantiate the inner class, with binding to the instance */  
        EnclosingClass.InnerClass innerClassInstance =  
            enclosingClassInstance.new InnerClass();  
  
        for(int i = enclosingClassInstance.getCounter(); (i =  
            innerClassInstance.incrementAndReturnCounter()) < 10;) {  
            System.out.println(i);  
        }  
    }  
}
```

Upon execution, this will print the integers from 0 to 9. Beware to not confuse this type of class with the so called static inner class, which is declared in the same way with an accompanied usage of the "static" modifier; those have not the desired effect but are instead just classes with no special binding defined in an enclosing class.

There have been a number of proposals for adding more fully featured closures to Java.^{[16][17][18]}

Blocks (C, C++, Objective-C 2.0)

Apple introduced **Blocks**, a form of closure, as a nonstandard extension into C, C++, Objective-C 2.0 and in Mac OS X 10.6 "Snow Leopard" and iOS 4.0.

Pointers to block and block literals are marked with `^`. Normal local variables are captured by value when the block is created, and are read-only inside the block. Variables to be captured by reference are marked with `__block`. Blocks that need to persist outside of the scope it is created in need to be copied.^{[19][20]}

```
typedef int (^IntBlock)();

IntBlock downCounter(int start) {
    __block int i = start;
    return [ [ ^int() {
        return i--;
    } copy] autorelease];
}

IntBlock f = downCounter(5);
NSLog(@"%@", f());
NSLog(@"%@", f());
NSLog(@"%@", f());
```

Delegates (C#, D)

C# anonymous methods and lambda expressions support closure to local variables:

```
var data = new[] {1, 2, 3, 4};
var multiplier = 2;
var result = data.Select(x => x * multiplier);
```

Closures are implemented by delegates in D.

```
auto test1() {
    int a = 7;
    return delegate() { return a + 3; }; // anonymous delegate
construction
}

auto test2() {
    int a = 20;
    int foo() { return a + 5; } // inner function
    return &foo; // other way to construct delegate
}

void bar() {
    auto dg = test1();
```

```

dg();      // =10  // ok, test1.a is in a closure and still exists

dg = test2();
dg();      // =25  // ok, test2.a is in a closure and still exists
}

```

D version 1, has limited closure support. For example, the above code will not work correctly, because the variable a is on the stack, and after returning from test(), it is no longer valid to use it (most probably calling foo via dg(), will return a 'random' integer). This can be solved by explicitly allocating the variable 'a' on heap, or using structs or class to store all needed closed variables and construct a delegate from a method implementing the same code. Closures can be passed to other functions, as long as they are only used while the referenced values are still valid (for example calling another function with a closure as a callback parameter), and are useful for writing generic data processing code, so this limitation, in practice, is often not an issue.

This limitation was fixed in D version 2 - the variable 'a' will be automatically allocated on the heap because it is used in the inner function, and a delegate of that function is allowed to escape the current scope (via assignment to dg or return). Any other local variables (or arguments) that are not referenced by delegates or that are only referenced by delegates that don't escape the current scope, remain on the stack, which is simpler and faster than heap allocation. The same is true for inner's class methods that references a function's variables.

Function objects (C++)

C++ allows defining function objects by overloading `operator()`. These objects behave somewhat like functions in a functional programming language. They may be created at runtime and may contain state, but they do not implicitly capture local variables as closures do. As of the 2011 revision, the C++ language also supports closures, which are a type of function object constructed automatically from a special language construct called *lambda-expression*. A C++ closure may capture its context either by storing copies of the accessed variables as members of the closure object or by reference. In the latter case, if the closure object escapes the scope of a referenced object, invoking its `operator()` causes undefined behavior since C++ closures do not extend the lifetime of their context.

```

void foo(string myname) {
    int y;
    vector<string> n;
    // ...
    auto i = std::find_if(n.begin(), n.end(), [&] (const string& s) {
        return s != myname && s.size() > y;
    });
    // 'i' is now either 'n.end()' or points to the first string in 'n'
    // which is not equal to 'myname' and whose length is greater than
    'y'
}

```

Inline agents (Eiffel)

Eiffel includes **inline agents** defining closures. An inline agent is an object representing a routine, defined by giving the code of the routine in-line. For example, in

```
ok_button.click_event.subscribe (
    agent (x, y: INTEGER) do
        map.country_at_coordinates (x, y).display
    end
)
```

the argument to `subscribe` is an agent, representing a procedure with two arguments; the procedure finds the country at the corresponding coordinates and displays it. The whole agent is "subscribed" to the event type `click_event` for a certain button, so that whenever an instance of the event type occurs on that button — because a user has clicked the button — the procedure will be executed with the mouse coordinates being passed as arguments for `x` and `y`.

The main limitation of Eiffel agents, which distinguishes them from true closures, is that they cannot reference local variables from the enclosing scope. Only `Current` (a reference to current object, analogous to `this` in Java), its features, and arguments of the agent itself can be accessed from within the agent body. This limitation is worked around by providing additional closed operands to the agent.

References

- [1] Sussman and Steele. "Scheme: An interpreter for extended lambda calculus". "... a data structure containing a lambda expression, and an environment to be used when that lambda expression is applied to arguments." (Wikisource)
- [2] Closures in Java (http://www.jugpadova.it/files/Closures_in_Java.pdf)
- [3] OpenJDK: Closures for the Java Programming Language (<http://openjdk.java.net/projects/closures/>), Project Lambda (<http://openjdk.java.net/projects/lambda/>); Closures (Lambda Expressions) for the Java Programming Language (<http://javac.info/>); James Gosling. "Closures" (<http://blogs.oracle.com/jag/entry/closures>); Guy Steele. Re: bindings and assignments (<http://article.gmane.org/gmane.comp.lang.lightweight/2274>).
- [4] P. J. Landin (1964), *The mechanical evaluation of expressions*
- [5] Joel Moses (June 1970) (PDF), *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should Be Called the Environment Problem* (<http://dspace.mit.edu/handle/1721.1/5854>), AI Memo 199, , retrieved 2009-10-27, "A useful metaphor for the difference between FUNCTION and QUOTE in LISP is to think of QUOTE as a porous or an open covering of the function since free variables escape to the current environment. FUNCTION acts as a closed or nonporous covering (hence the term "closure" used by Landin). Thus we talk of "open" Lambda expressions (functions in LISP are usually Lambda expressions) and "closed" Lambda expressions. [...] My interest in the environment problem began while Landin, who had a deep understanding of the problem, visited MIT during 1966-67. I then realized the correspondence between the FUNARG lists which are the results of the evaluation of "closed" Lambda expressions in LISP and ISWIM's Lambda Closures."
- [6] Åke Wikström (1987). *Functional Programming using Standard ML*. ISBN 0-13-331968-7. "The reason it is called a "closure" is that an expression containing free variables is called an "open" expression, and by associating to it the bindings of its free variables, you close it."
- [7] Gerald Jay Sussman and Guy L. Steele, Jr. (December 1975), *Scheme: An Interpreter for the Extended Lambda Calculus*, AI Memo 349
- [8] *Lambda Expressions and Closures* (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2008/n2550.pdf>) C++ Standards Committee. 29 February 2008.
- [9] *Foundations of Actor Semantics* (<https://dspace.mit.edu/handle/1721.1/6935>) Will Clinger. MIT Mathematics Doctoral Dissertation. June 1981.
- [10] "array.filter" (https://developer.mozilla.org/en/Core_JavaScript_1.5_Reference/Global_Objects/Array/filter). *Mozilla Developer Center*. 10 January 2010.. Retrieved 2010-02-09.
- [11] "Re: FP, OO and relations. Does anyone trump the others?" (<http://okmij.org/ftp/Scheme/oop-in-fp.txt>). 29 December 1999.. Retrieved 2008-12-23.
- [12] "State of the Lambda" (<http://cr.openjdk.java.net/~briangoetz/lambda/lambda-state-4.html>) .
- [13] "Nested, Inner, Member, and Top-Level Classes" (https://blogs.oracle.com/darcy/entry/nested_inner_member_and_top) .
- [14] "Inner Class Example (The Java Tutorials > Learning the Java Language > Classes and Objects)" (<http://java.sun.com/docs/books/tutorial/java/javaOO/innerclasses.html>) .
- [15] "Nested Classes (The Java Tutorials > Learning the Java Language > Classes and Objects)" (<http://java.sun.com/docs/books/tutorial/java/javaOO/nested.html>) .

- [16] <http://www.javac.info/>
- [17] http://docs.google.com/View?docid=k73_1ggr36h
- [18] http://docs.google.com/Doc?id=ddhp95vd_0f7mcns
- [19] Apple Inc.. "Blocks Programming Topics" (http://developer.apple.com/library/mac/#documentation/Cocoa/Conceptual/Blocks/Articles/00_Introduction.html). . Retrieved 2011-03-08.
- [20] Joachim Bengtsson (7 July 2010). "Programming with C Blocks On Apple Devices" (<http://thirdcog.eu/pwcblocks/>). . Retrieved 2010-09-18.

External links

- The Original "Lambda Papers" (<http://library.readscheme.org/page1.html>): A classic series of papers by Guy Steele and Gerald Sussman discussing, among other things, the versatility of closures in the context of Scheme (where they appear as *lambda expressions*).
- Neal Gafter (2007-01-28). "A Definition of Closures" (<http://gafter.blogspot.com/2007/01/definition-of-closures.html>).
- Gilad Bracha, Neal Gafter, James Gosling, Peter von der Ahé. "Closures for the Java Programming Language (v0.5)" (<http://www.javac.info/closures-v05.html>).
- Closures (<http://martinfowler.com/bliki/Closure.html>): An article about closures in dynamically typed imperative languages, by Martin Fowler.
- Collection closure methods (<http://martinfowler.com/bliki/CollectionClosureMethod.html>): An example of a technical domain where using closures is convenient, by Martin Fowler.

Javascript

- What are closures (<http://blogs.msdn.com/kartikb/archive/2009/02/08/closures.aspx>): A post on closures in Javascript.
- Javascript Closures for Dummies (http://web.archive.org/web/20080209105120/http://blog.morrisjohns.com/javascript_closures_for_dummies): An article teaching closures in Javascript by examples.

Java and .NET

- The beauty of closures (<http://www.developerfusion.com/article/8251/the-beauty-of-closures/>): An article about using closures in Java and .NET

Delphi

- Nick Hodges, " Delphi 2009 Reviewers Guide (<http://dn.codegear.com/article/38757>)", October 2008, *CodeGear Developer Network*, CodeGear.
- Craig Stuntz, " Understanding Anonymous Methods (<http://blogs.teamb.com/craigstuntz/2008/08/04/37828/>)", October 2008
- Dr. Bob, " Delphi 2009 Anonymous Methods (<http://www.drbob42.com/examines/examinA5.htm>)"

Ruby

- Robert Sosinski, " Understanding Ruby Blocks, Procs and Lambdas (<http://www.robertsosinski.com/2008/12/21/understanding-ruby-blocks-procs-and-lambdas/>)", December 2008

Functional programming

In computer science, **functional programming** is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids state and mutable data. It emphasizes the application of functions, in contrast to the imperative programming style, which emphasizes changes in state.^[1] Functional programming has its roots in lambda calculus, a formal system developed in the 1930s to investigate function definition, function application, and recursion. Many functional programming languages can be viewed as elaborations on the lambda calculus.^[1]

In practice, the difference between a mathematical function and the notion of a function used in imperative programming is that imperative functions can have side effects that may change the value of program state. Because of this, they lack referential transparency, i.e. the same language expression can result in different values at different times depending on the state of the executing program. Conversely, in functional code, the output value of a function depends only on the arguments that are input to the function, so calling a function f twice with the same value for an argument x will produce the same result $f(x)$ both times. Eliminating side effects can make it much easier to understand and predict the behavior of a program, which is one of the key motivations for the development of functional programming.^[1]

Functional programming languages, especially purely functional ones such as the pioneering Hope, have largely been emphasized in academia rather than in commercial software development. However, prominent functional programming languages such as Common Lisp, Scheme,^{[2][3][4][5]} ISLISP, Clojure, Racket,^[6] Erlang,^{[7][8][9]} OCaml,^{[10][11]} Haskell,^{[12][13]} Scala^[14] and F#^{[15][16]} have been used in industrial and commercial applications by a wide variety of organizations. Functional programming is also supported in some domain-specific programming languages like R (statistics),^{[17][18]} Mathematica (symbolic math),^[19] J, K and Q from Kx Systems (financial analysis), XQuery/XSLT (XML)^{[20][21]} and Opal.^[1] Widespread domain-specific declarative languages like SQL and Lex/Yacc use some elements of functional programming, especially in eschewing mutable values.^[22]

Programming in a functional style can also be accomplished in languages that aren't specifically designed for functional programming. For example, the imperative Perl programming language has been the subject of a book describing how to apply functional programming concepts.^[23] C# 3.0 added constructs to facilitate the functional style as well and even Apple's Objective-c uses blocks to provide a functional support.

History

Lambda calculus provides a theoretical framework for describing functions and their evaluation. Although it is a mathematical abstraction rather than a programming language, it forms the basis of almost all functional programming languages today. An equivalent theoretical formulation, combinatory logic, is commonly perceived as more abstract than lambda calculus and preceded it in invention. It is used in some esoteric languages including Unlambda. Combinatory logic and lambda calculus were both originally developed to achieve a clearer approach to the foundations of mathematics.^[24]

An early functional-flavored language was Lisp, developed by John McCarthy while at Massachusetts Institute of Technology (MIT) for the IBM 700/7000 series scientific computers in the late 1950s.^[25] Lisp introduced many features now found in functional languages, though Lisp is technically a multi-paradigm language. Scheme and Dylan were later attempts to simplify and improve Lisp.

Information Processing Language (IPL) is sometimes cited as the first computer-based functional programming language.^[26] It is an assembly-style language for manipulating lists of symbols. It does have a notion of "generator", which amounts to a function accepting a function as an argument, and, since it is an assembly-level language, code can be used as data, so IPL can be regarded as having higher-order functions. However, it relies heavily on mutating list structure and similar imperative features.

Kenneth E. Iverson developed APL in the early 1960s, described in his 1962 book *A Programming Language* (ISBN 9780471430148). APL was the primary influence on John Backus's FP. In the early 1990s, Iverson and Roger Hui created J. In the mid 1990s, Arthur Whitney, who had previously worked with Iverson, created K, which is used commercially in financial industries along with its descendant Q.

John Backus presented FP in his 1977 Turing Award lecture "Can Programming Be Liberated From the von Neumann Style? A Functional Style and its Algebra of Programs".^[27] He defines functional programs as being built up in a hierarchical way by means of "combining forms" that allow an "algebra of programs"; in modern language, this means that functional programs follow the principle of compositionality. Backus's paper popularized research into functional programming, though it emphasized function-level programming rather than the lambda-calculus style which has come to be associated with functional programming.

In the 1970s, ML was created by Robin Milner at the University of Edinburgh, and David Turner developed initially the language SASL at the University of St. Andrews and later the language Miranda at the University of Kent. ML eventually developed into several dialects, the most common of which are now OCaml and Standard ML. Also in the 1970s, the development of Scheme (a partly functional dialect of Lisp), as described in the influential Lambda Papers and the 1985 textbook *Structure and Interpretation of Computer Programs*, brought awareness of the power of functional programming to the wider programming-languages community.

In the 1980s, Per Martin-Löf developed intuitionistic type theory (also called *constructive* type theory), which associated functional programs with constructive proofs of arbitrarily complex mathematical propositions expressed as dependent types. This led to powerful new approaches to interactive theorem proving and has influenced the development of many subsequent functional programming languages.

The Haskell language began with a consensus in 1987 to form an open standard for functional programming research; implementation releases have been ongoing since 1990.

Concepts

A number of concepts and paradigms are specific to functional programming, and generally foreign to imperative programming (including object-oriented programming). However, programming languages are often hybrids of several programming paradigms, so programmers using "mostly imperative" languages may have utilized some of these concepts.^[28]

First-class and higher-order functions

Higher-order functions are functions that can either take other functions as arguments or return them as results. In calculus, an example of a higher-order function is the differential operator d/dx , which returns the derivative of a function f .

Higher-order functions are closely related to first-class functions in that higher-order functions and first-class functions both allow functions as arguments and results of other functions. The distinction between the two is subtle: "higher-order" describes a mathematical concept of functions that operate on other functions, while "first-class" is a computer science term that describes programming language entities that have no restriction on their use (thus first-class functions can appear anywhere in the program that other first-class entities like numbers can, including as arguments to other functions and as their return values).

Higher-order functions enable partial application or currying, a technique in which a function is applied to its arguments one at a time, with each application returning a new function that accepts the next argument. This allows one to succinctly express, for example, the successor function as the addition operator partially applied to the natural number one.

Pure functions

Purely functional functions (or expressions) have no memory or I/O side effects. This means that pure functions have several useful properties, many of which can be used to optimize the code:

- If the result of a pure expression is not used, it can be removed without affecting other expressions.
- If a pure function is called with parameters that cause no side-effects, the result is constant with respect to that parameter list (sometimes called referential transparency), i.e. if the pure function is again called with the same parameters, the same result will be returned (this can enable caching optimizations such as memoization).
- If there is no data dependency between two pure expressions, then their order can be reversed, or they can be performed in parallel and they cannot interfere with one another (in other terms, the evaluation of any pure expression is thread-safe).
- If the entire language does not allow side-effects, then any evaluation strategy can be used; this gives the compiler freedom to reorder or combine the evaluation of expressions in a program (for example, using deforestation).

While most compilers for imperative programming languages detect pure functions and perform common-subexpression elimination for pure function calls, they cannot always do this for pre-compiled libraries, which generally do not expose this information, thus preventing optimizations that involve those external functions. Some compilers, such as gcc, add extra keywords for a programmer to explicitly mark external functions as pure, to enable such optimizations. Fortran 95 also allows functions to be designated "pure".

Recursion

Iteration (looping) in functional languages is usually accomplished via recursion. Recursive functions invoke themselves, allowing an operation to be performed over and over. Recursion may require maintaining a stack, but tail recursion can be recognized and optimized by a compiler into the same code used to implement iteration in imperative languages. The Scheme language standard requires implementations to recognize and optimize tail recursion. Tail recursion optimization can be implemented by transforming the program into continuation passing style during compiling, among other approaches.

Common patterns of recursion can be factored out using higher order functions, with catamorphisms and anamorphisms (or "folds" and "unfolds") being the most obvious examples. Such higher order functions play a role analogous to built-in control structures such as loops in imperative languages.

Most general purpose functional programming languages allow unrestricted recursion and are Turing complete, which makes the halting problem undecidable, can cause unsoundness of equational reasoning, and generally requires the introduction of inconsistency into the logic expressed by the language's type system. Some special purpose languages such as Coq allow only well-founded recursion and are strongly normalizing (nonterminating computations can be expressed only with infinite streams of values called codata). As a consequence, these languages fail to be Turing complete and expressing certain functions in them is impossible, but they can still express a wide class of interesting computations while avoiding the problems introduced by unrestricted recursion. Functional programming limited to well-founded recursion with a few other constraints is called total functional programming. See Turner 2004 for more discussion.^[29]

Strict versus non-strict evaluation

Functional languages can be categorized by whether they use *strict* (*eager*) or *non-strict* (*lazy*) evaluation, concepts that refer to how function arguments are processed when an expression is being evaluated. The technical difference is in the denotational semantics of expressions containing failing or divergent computations. Under strict evaluation, the evaluation of any term containing a failing subterm will itself fail. For example, the expression:

```
print length([2+1, 3*2, 1/0, 5-4])
```

will fail under strict evaluation because of the division by zero in the third element of the list. Under nonstrict evaluation, the length function will return the value 4 (i.e., the number of items in the list), since evaluating it will not attempt to evaluate the terms making up the list. In brief, strict evaluation always fully evaluates function arguments before invoking the function. Non-strict evaluation does not evaluate function arguments unless their values are required to evaluate the function call itself.

The usual implementation strategy for non-strict evaluation in functional languages is graph reduction.^[30] Non-strict evaluation is used by default in several pure functional languages, including Miranda, Clean, and Haskell.

Hughes 1984 argues for non-strict evaluation as a mechanism for improving program modularity through separation of concerns, by easing independent implementation of producers and consumers of data streams.^[31] Launchbury 1993 describes some difficulties that lazy evaluation introduces, particularly in analyzing a program's storage requirements, and proposes an operational semantics to aid in such analysis.^[32] Harper 2009 proposes including both strict and nonstrict evaluation in the same language, using the language's type system to distinguish them.^[33]

Type systems

Especially since the development of Hindley–Milner type inference in the 1970s, functional programming languages have tended to use typed lambda calculus, as opposed to the untyped lambda calculus used in Lisp and its variants (such as Scheme). The use of algebraic datatypes and pattern matching makes manipulation of complex data structures convenient and expressive; the presence of strong compile-time type checking makes programs more reliable, while type inference frees the programmer from the need to manually declare types to the compiler.

Some research-oriented functional languages such as Coq, Agda, Cayenne, and Epigram are based on intuitionistic type theory, which allows types to depend on terms. Such types are called dependent types. These type systems do not have decidable type inference and are difficult to understand and program with. But dependent types can express arbitrary propositions in predicate logic. Through the Curry–Howard isomorphism, then, well-typed programs in these languages become a means of writing formal mathematical proofs from which a compiler can generate certified code. While these languages are mainly of interest in academic research (including in formalized mathematics), they have begun to be used in engineering as well. CompCert is a compiler for a subset of the C programming language that is written in Coq and formally verified.^[34]

A limited form of dependent types called generalized algebraic data types (GADT's) can be implemented in a way that provides some of the benefits of dependently typed programming while avoiding most of its inconvenience.^[35] GADT's are available in the Glasgow Haskell Compiler and in Scala (as "case classes"), and have been proposed as additions to other languages including Java and C#.^[36]

Functional programming in non-functional languages

It is possible to use a functional style of programming in languages that are not traditionally considered functional languages.^[37] For example, both D and Fortran 95 explicitly support pure functions.^[38]

First class functions have slowly been added to mainstream languages. For example, in early 1994, support for lambda, filter, map, and reduce was added to Python. Then during the development of Python 3000, Guido van Rossum called for the removal of these features.^{[39][40]} So far, only the `reduce` function has been removed, and it remains accessible via the `functools` standard library module.^[41] First class functions were also introduced in PHP 5.3, Visual Basic 9, C# 3.0, and C++11.

The Language Integrated Query (LINQ) feature, with its many incarnations, is an obvious and powerful use of functional programming in .NET.

In Java, anonymous classes can sometimes be used to simulate closures; however, anonymous classes are not always proper replacements to closures because they have more limited capabilities.

Many object-oriented design patterns are expressible in functional programming terms: for example, the strategy pattern simply dictates use of a higher-order function, and the visitor pattern roughly corresponds to a catamorphism, or fold.

Similarly, the idea of immutable data from functional programming is often included in imperative programming languages,^[42] for example the tuple in Python, which is an immutable array.

Comparison to imperative programming

Functional programming is very different from imperative programming. The most significant differences stem from the fact that functional programming avoids side effects, which are used in imperative programming to implement state and I/O. Pure functional programming disallows side effects completely and so provides referential transparency, which makes it easier to verify, optimize, and parallelize programs, and easier to write automated tools to perform those tasks.

Higher-order functions are rarely used in older imperative programming. Where a traditional imperative program might use a loop to traverse a list, a functional program would use a different technique. It would use a higher-order function that takes as arguments a function and a list. The higher-order function would then apply the given function to each element of the given list and then return a new list with the results.

Simulating state

There are tasks (for example, maintaining a bank account balance) that often seem most naturally implemented with state. Pure functional programming performs these tasks, and I/O tasks such as accepting user input and printing to the screen, in a different way.

The pure functional programming language Haskell implements them using monads, derived from category theory. Monads offer a way to abstract certain types of computational patterns, including (but not limited to) modeling of computations with mutable state (and other side effects such as I/O) in an imperative manner without losing purity. While existing monads may be easy to apply in a program, given appropriate templates and examples, many students find them difficult to understand conceptually, e.g., when asked to define new monads (which is sometimes needed for certain types of libraries).^[43]

Another way in which functional languages can simulate state is by passing around a data structure that represents the current state as a parameter to function calls. On each function call, a copy of this data structure is created with whatever differences are the result of the function. This is referred to as 'state-passing style'.

Impure functional languages usually include a more direct method of managing mutable state. Clojure, for example, uses managed references that can be updated by applying pure functions to the current state. This kind of approach

enables mutability while still promoting the use of pure functions as the preferred way to express computations.

Alternative methods such as Hoare logic and uniqueness have been developed to track side effects in programs. Some modern research languages use effect systems to make explicit the presence of side effects.

Efficiency issues

Functional programming languages are typically less efficient in their use of CPU and memory than imperative languages such as C and Pascal.^[44] This is related to the fact that some mutable data structures like arrays have a very straightforward implementation using present hardware (which is a highly evolved Turing machine). Flat arrays may be accessed very efficiently with deeply pipelined CPUs, prefetched efficiently through caches (with no complex pointer-chasing), or handled with SIMD instructions. And it is not easy to create their equally efficient general-purpose immutable counterparts. For purely functional languages, the worst-case slowdown is logarithmic in the number of memory cells used, because mutable memory can be represented by a purely functional data structure with logarithmic access time (such as a balanced tree).^[45] However, such slowdowns are not universal. For programs that perform intensive numerical computations, functional languages such as OCaml and Clean are only slightly slower than C.^[46] For programs that handle large matrices and multidimensional databases, array functional languages (such as J and K) were designed with speed optimization.

Immutability of data can, in many cases, lead to execution efficiency, by allowing the compiler to make assumptions that are unsafe in an imperative language, thus increasing opportunities for inline expansion.

Lazy evaluation may also speed up the program, even asymptotically, whereas it may slow it down at most by a constant factor (however, it may introduce memory leaks when used improperly). Launchbury 1993^[32] discusses theoretical issues related to memory leaks from lazy evaluation, and O'Sullivan *et al.* 2008^[47] give some practical advice for analyzing and fixing them. However, the most general implementations of lazy evaluation making extensive use of dereferenced code and data perform poorly on modern processors with deep pipelines and multi-level caches (where a cache miss may cost hundreds of cycles)

Coding styles

Imperative programs tend to emphasize the series of steps taken by a program in carrying out an action, while functional programs tend to emphasize the composition and arrangement of functions, often without specifying explicit *steps*. A simple example illustrates this with two solutions to the same programming goal (calculating Fibonacci numbers). The imperative example is in C++.

```
#include <iostream>

// Fibonacci numbers, imperative style
int fibonacci(int iterations)
{
    int first = 0, second = 1; // seed values

    for (int i = 0; i < iterations - 1; ++i) {
        int sum = first + second;
        first = second;
        second = sum;
    }

    return first;
}
```

```
int main()
{
    std::cout << fibonacci(10) << "\n";
    return 0;
}
```

Haskell

A functional version (in Haskell) has a different feel to it:

```
-- Fibonacci numbers, functional style

-- describe an infinite list based on the recurrence relation for
Fibonacci numbers
fibRecurrence first second = first : fibRecurrence second (first +
second)

-- describe fibonacci list as fibRecurrence with initial values 0 and 1
fibonacci = fibRecurrence 0 1

-- describe action to print the 10th element of the fibonacci list
main = print (fibonacci !! 10)
```

Or, more concisely:

```
fibonacci2 = 0:1:zipwith (+) fibonacci2 (tail fibonacci2)
```

The imperative style describes the intermediate steps involved in calculating `fibonacci(N)`, and places those steps inside a loop statement. In contrast, the functional implementation shown here states the mathematical recurrence relation that defines the entire Fibonacci sequence, then selects an element from the sequence (see also recursion). This example relies on Haskell's lazy evaluation to create an "infinite" list of which only as much as needed (the first 10 elements in this case) will actually be computed. That computation happens when the runtime system carries out the action described by "main".

Erlang

The same program in Erlang provides a simple example of how functional languages in general do not require their syntax to contain an "if" statement.

```
-module(fibonacci).
-export([start/1]).

%% Fibonacci numbers in Erlang
start(N) -> do_fib(0,1,N).

do_fib(_,B,1) -> B;
do_fib(A,B,N) -> do_fib(B,A+B,N-1).
```

This program is contained within a module called "fibonacci" and declares that the `start/1` function will be visible from outside the scope of this module.

The function `start/1` accepts a single parameter (as denoted by the "/1" syntax) and then calls an internal function called `do_fib/3`.

In direct contrast to the imperative coding style, Erlang does not need an "if" statement because the Erlang runtime will examine the parameters being passed to a function, and call the first function having a signature that matches the current pattern of parameters. (Erlang syntax does provide an "if" statement, but it is considered syntactic sugar and, compared to its usage in imperative languages, plays only a minor role in application logic design).

In this case, it is unnecessary to test for a parameter value within the body of the function because such a test is implicitly performed by providing a set of function signatures that describe the different patterns of values that could be received by a function.

In the case above, the first version of do_fib/3 will only be called when the third parameter has the precise value of 1. In all other cases, the second version of do_fib/3 will be called.

This example demonstrates that functional programming languages often implement conditional logic *implicitly* by matching parameter patterns rather than *explicitly* by means of an "if" statement.

Lisp

Writing a program to compute a factorial, a similarly recursive mathematical problem, in Lisp results in:

```
(defun fact (x) (if (= x 1) 1 (* x (fact (- x 1)))))
```

The program can then be called as

```
(fact 10)
```

SML

A fibonacci function written in SML using pattern matching bears a direct resemblance to its formal mathematical definition:

```
fun fibonacci 0 = 1
| fibonacci 1 = 1
| fibonacci n = fibonacci(n-1)+fibonacci(n-2);
```

In a SML interactive interpreter the function can be called as

```
fibonacci 15;
```

This form allows a mathematician to turn a mathematical function into a computer function with little training in the syntax of the computer language.

Use in industry

Functional programming has long been popular in academia, but with few industrial applications.^{[48]:page 11} However, recently several prominent functional programming languages have been used in commercial or industrial systems. For example, the Erlang programming language, which was developed by the Swedish company Ericsson in the late 1980s, was originally used to implement fault-tolerant telecommunications systems.^[8] It has since become popular for building a range of applications at companies such as T-Mobile, Nortel, Facebook and EDF.^{[7][9][49][50]} The Scheme dialect of Lisp was used as the basis for several applications on early Apple Macintosh computers,^{[2][3]} and has more recently been applied to problems such as training simulation software^[4] and telescope control.^[5] OCaml, which was introduced in the mid 1990s, has seen commercial use in areas such as financial analysis,^[10] driver verification, industrial robot programming, and static analysis of embedded software.^[11] Haskell, although initially intended as a research language,^[13] has also been applied by a range of companies, in areas such as aerospace systems, hardware design, and web programming.^{[12][13]}

Other functional programming languages that have seen use in industry include Scala,^[51] F#,^{[15][16]} Lisp,^[52] Standard ML,^{[53][54]} and Clojure.^[55]

References

- [1] Hudak, Paul (September 1989). "Conception, evolution, and application of functional programming languages" (<http://www.cs.berkeley.edu/~jcondit/pl-prelim/hudak89functional.pdf>) (PDF). *ACM Computing Surveys* **21** (3): 359–411. doi:10.1145/72551.72554. .
- [2] Clinger, Will (1987). "MultiTasking and MacScheme" (<http://www.mactech.com/articles/mactech/Vol.03/03.12/Multitasking/index.html>). *MacTech* **3** (12). . Retrieved 2008-08-28.
- [3] Hartheimer, Anne (1987). "Programming a Text Editor in MacScheme+Toolsmith" (<http://www.mactech.com/articles/mactech/Vol.03/03.1/SchemeWindows/index.html>). *MacTech* **3** (1). . Retrieved 2008-08-28.
- [4] Kidd, Eric. "Terrorism Response Training in Scheme" (<http://cufp.galois.com/2007/abstracts.html#EricKidd>). CUFP 2007. . Retrieved 2009-08-26.
- [5] Cleis, Richard. "Scheme in Space" (<http://cufp.galois.com/2006/abstracts.html#RichardCleis>). CUFP 2006. . Retrieved 2009-08-26.
- [6] "State-Based Scripting in Uncharted 2" (<http://www.gameenginebook.com/gdc09-statescripting-charted2.pdf>). . Retrieved 2011-08-08.
- [7] "Who uses Erlang for product development?" (<http://www.erlang.org/faq/faq.html#AEN50>). *Frequently asked questions about Erlang*. . Retrieved 2007-08-05.
- [8] Armstrong, Joe (June 2007). "A history of Erlang" (<http://doi.acm.org/10.1145/1238844.1238850>). Third ACM SIGPLAN Conference on History of Programming Languages. San Diego, California. . Retrieved 2009-08-29.
- [9] Larson, Jim (March 2009). "Erlang for concurrent programming" (<http://doi.acm.org/10.1145/1467247.1467263>). *Communications of the ACM* **52** (3): 48. doi:10.1145/1467247.1467263. . Retrieved 2009-08-29.
- [10] Minsky, Yaron; Weeks, Stephen (July 2008). "Caml Trading — experiences with functional programming on Wall Street" (<http://journals.cambridge.org/action/displayAbstract?aid=1899164>). *Journal of Functional Programming* (Cambridge University Press) **18** (4): 553–564. doi:10.1017/S095679680800676X. . Retrieved 2008-08-27.
- [11] Leroy, Xavier. "Some uses of Caml in Industry" (<http://cufp.galois.com/2007/slides/XavierLeroy.pdf>). CUFP 2007. . Retrieved 2009-08-26.
- [12] "Haskell in industry" (http://www.haskell.org/haskellwiki/Haskell_in_industry). *Haskell Wiki*. . Retrieved 2009-08-26. "Haskell has a diverse range of use commercially, from aerospace and defense, to finance, to web startups, hardware design firms and lawnmower manufacturers."
- [13] Hudak, Paul; Hughes, J., Jones, S. P., and Wadler, P. (June 2007). "A history of Haskell: being lazy with class" (<http://doi.acm.org/10.1145/1238844.1238856>). Third ACM SIGPLAN Conference on History of Programming Languages. San Diego, California. . Retrieved 2009-08-29.
- [14] "Effective Scala" (<http://twitter.github.com/effectivescala/?sd>). *Scala Wiki*. . Retrieved 2012-02-21. "Effective Scala."
- [15] Mansell, Howard (2008). "Quantitative Finance in F#" (<http://cufp.galois.com/2008/abstracts.html#MansellHoward>). CUFP 2008. . Retrieved 2009-08-29.
- [16] Peake, Alex (2009). "The First Substantial Line of Business Application in F#" (<http://cufp.galois.com/2009/abstracts.html#AlexPeakeAdamGranicz>). CUFP 2009. . Retrieved 2009-08-29.
- [17] "The useR! 2006 conference schedule includes papers on the commercial use of R" (<http://www.r-project.org/useR-2006/program.html>). R-project.org. 2006-06-08. . Retrieved 2011-06-20.
- [18] Chambers, John M. (1998). *Programming with Data: A Guide to the S Language*. Springer Verlag. pp. 67–70. ISBN 978-0-387-98503-9.
- [19] Department of Applied Math, University of Colorado. "Functional vs. Procedural Programming Language" (<http://amath.colorado.edu/computing/mmm/funcproc.html>). . Retrieved 2006-08-28.
- [20] Dimitre Novatchev. "The Functional Programming Language XSLT — A proof through examples" (<http://www.topxml.com/xsl/articles/fp/>). *TopXML*. . Retrieved May 27, 2006.
- [21] David Mertz. "XML Programming Paradigms (part four): Functional Programming approached to XML processing" (http://gnosis.cx/publish/programming/xml_models_fp.html). *IBM developerWorks*. . Retrieved May 27, 2006.
- [22] Donald D. Chamberlin and Raymond F. Boyce (1974). "SEQUEL: A structured English query language". *Proceedings of the 1974 ACM SIGFIDET*: 249–264.
- [23] Dominus, Mark J. (2005). *Higher-Order Perl*. Morgan Kaufmann. ISBN 1-55860-701-3.
- [24] Curry, Haskell Brooks; Robert Feys and Craig, William (1958). *Combinatory Logic. Volume I*. Amsterdam: North-Holland Publishing Company.
- [25] McCarthy, John (June 1978). "History of Lisp" (<http://citeseer.ist.psu.edu/mccarthy78history.html>). In *ACM SIGPLAN History of Programming Languages Conference*: 217–223. doi:10.1145/800025.808387. .
- [26] The memoir of Herbert A. Simon (1991), *Models of My Life* pp.189-190 ISBN 0-465-04640-1 claims that he, Al Newell, and Cliff Shaw are "commonly adjudged to be the parents of [the] artificial intelligence [field]", for writing Logic Theorist, a program which proved theorems from *Principia Mathematica* automatically. In order to accomplish this, they had to invent a language and a paradigm which, which viewed retrospectively, embeds functional programming.
- [27] <http://www.stanford.edu/class/cs242/readings/backus.pdf>
- [28] Dick Pountain. "Functional Programming Comes of Age" (<http://www.byte.com/art/9408/sec11/art1.htm>). *BYTE.com* (August 1994). . Retrieved August 31, 2006.
- [29] Turner, D.A. (2004-07-28). "Total Functional Programming" (http://www.jucs.org/jucs_10_7/total_functional_programming). *Journal of Universal Computer Science* **10** (7): 751–768. doi:10.3217/jucs-010-07-0751.

- [30] The Implementation of Functional Programming Languages. Simon Peyton Jones, published by Prentice Hall, 1987 (<http://research.microsoft.com/~simonpj/papers/slpj-book-1987/index.htm>)
- [31] Hughes, John (1984). "Why Functional Programming Matters" (<http://www.cse.chalmers.se/~rjmh/Papers/whyfp.html>). .
- [32] John Launchbury (1993). "A Natural Semantics for Lazy Evaluation" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.2016>). .
- [33] Robert W. Harper (2009). *Practical Foundations for Programming Languages* (<http://www.cs.cmu.edu/~rwh/plbook/book.pdf>). .
- [34] "The Compcert verified compiler" (<http://compcert.inria.fr/doc/index.html>). .
- [35] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. "Simple unification-based type inference for GADTs" (<http://research.microsoft.com/en-us/um/people/simonpj/papers/gadt/>). *ICFP 2006*. pp. 50–61. .
- [36] Andrew Kennedy and Claudio Russo (October 2005). "Generalized Algebraic Data Types and Object-Oriented Programming" (<http://research.microsoft.com/~akenn/generics/gadtoop.pdf>). *OOPSLA*. San Diego, California. . source of citation (<http://lambda-the-ultimate.org/node/1134>)
- [37] Hartel, Pieter; Henk Muller and Hugh Glaser (March 2004). "The Functional C experience" (<http://www.ub.utwente.nl/webdocs/cit/1/00000084.pdf>) (PDF). *The Journal of Functional Programming* **14** (2): 129–135. doi:10.1017/S0956796803004817. . ; David Mertz. "Functional programming in Python, Part 3" (<http://www-128.ibm.com/developerworks/linux/library/l-prog3.html>). *IBM developerWorks*. . Retrieved 2006-09-17. (Part 1 (<http://www-128.ibm.com/developerworks/library/l-prog.html>), Part 2 (<http://www-128.ibm.com/developerworks/library/l-prog2.html>))
- [38] "Functions — D Programming Language 2.0" (<http://www.digitalmars.com/d/2.0/function.html#pure-functions>). Digital Mars. . Retrieved 2011-06-20.
- [39] van Rossum, Guido (2005-03-10). "The fate of reduce() in Python 3000" (<http://www.artima.com/weblogs/viewpost.jsp?thread=98196>). Artima.com. . Retrieved 2012-09-27.
- [40] van Rossum, Guido (2009-04-21). "Origins of Python's "Functional" Features" (<http://python-history.blogspot.de/2009/04/origins-of-pythons-functional-features.html>). The History of Python (<http://python-history.blogspot.de/>). . Retrieved 2012-09-27.
- [41] "functools — Higher order functions and operations on callable objects" (<http://docs.python.org/dev/library/functools.html#functools.reduce>). Python Software Foundation. 2011-07-31. . Retrieved 2011-07-31.
- [42] Bloch, Joshua. *Effective Java* (Second ed.). pp. Item 15.
- [43] Newbern, J.. "All About Monads: A comprehensive guide to the theory and practice of monadic programming in Haskell" (<http://monads.haskell.cz/html/index.html/html/>). . Retrieved 2008-02-14.
- [44] Lawrence C. Paulson, *ML for the Working Programmer*. Cambridge UP, 1996. ISBN 0-521-56543-X.
- [45] Paweł Spiewak. "Implementing Persistent Vectors in Scala" (<http://www.codecommit.com/blog/scala/implementing-persistent-vectors-in-scala>). . Retrieved Apr 17, 2012.
- [46] "Boxplot Summary | Computer Language Benchmarks Game" (<http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=all&d=data&calc=calculate&gcc=on&clean=on&sbcl=on&ocaml=on&ghc=on&fsharp=on&hipe=on&mzscheme=on>). Shootout.alioth.debian.org. . Retrieved 2011-06-20.
- [47] "Chapter 25. Profiling and optimization" (http://book.realworldhaskell.org/read/profiling-and-optimization.html#x_eK1). Book.realworldhaskell.org. . Retrieved 2011-06-20.
- [48] Odersky, Martin; Spoon, Lex; Venners, Bill (December 13, 2010). *Programming in Scala: A Comprehensive Step-by-step Guide* (http://www.artima.com/shop/programming_in_scala_2ed) (2nd ed.). Artima Inc. pp. 883/852. ISBN 978-0-9815316-4-9. .
- [49] Piro, Christopher (2009). "Functional Programming at Facebook" (<http://cufp.galois.com/2009/abstracts.html#ChristopherPiroEugeneLetuchy>). CUFN 2009. . Retrieved 2009-08-29.
- [50] "Sim-Diasca: a large-scale discrete event concurrent simulation engine in Erlang" (<http://research.edf.com/research-and-the-scientific-community/software/sim-diasca-80704.html>). November 2011. .
- [51] Momtahan, Lee (2009). "Scala at EDF Trading: Implementing a Domain-Specific Language for Derivative Pricing with Scala" (<http://cufp.galois.com/2009/abstracts.html#LeeMomtahan>). CUFN 2009. . Retrieved 2009-08-29.
- [52] Graham, Paul (2003). "Beating the Averages" (<http://www.paulgraham.com/avg.html>). . Retrieved 2009-08-29.
- [53] Sims, Steve (2006). "Building a Startup with Standard ML" (<http://cufp.galois.com/2006/slides/SteveSims.pdf>). CUFN 2006. . Retrieved 2009-08-29.
- [54] Laurikari, Ville (2007). "Functional Programming in Communications Security." (<http://cufp.galois.com/2007/abstracts.html#VilleLaurikari>). CUFN 2007. . Retrieved 2009-08-29.
- [55] Lorimer, R. J.. "Live Production Clojure Application Announced" (http://www.infoq.com/news/2009/01/clojure_production). .

Further reading

- Abelson, Hal; Sussman, Gerald Jay (1985). *[[Structure and Interpretation of Computer Programs* (<http://mitpress.mit.edu/sicp/>)]]. MIT Press.
- Cousineau, Guy and Michel Mauny. *The Functional Approach to Programming*. Cambridge, UK: Cambridge University Press, 1998.
- Curry, Haskell Brooks and Feys, Robert and Craig, William. *Combinatory Logic*. Volume I. North-Holland Publishing Company, Amsterdam, 1958.
- Curry, Haskell B.; Hindley, J. Roger; Seldin, Jonathan P. (1972). *Combinatory Logic*. Vol. II. Amsterdam: North Holland. ISBN 0-7204-2208-6.
- Dominus, Mark Jason. *Higher-Order Perl*. Morgan Kaufmann. 2005.
- Felleisen, Matthias; Findler, Robert; Flatt, Matthew; Krishnamurthi, Shriram (2001). *[[How to Design Programs* (<http://www.htdp.org/>)]]. MIT Press.
- Graham, Paul. *ANSI Common LISP*. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
- MacLennan, Bruce J. *Functional Programming: Practice and Theory*. Addison-Wesley, 1990.
- O'Sullivan, Brian; Stewart, Don; Goerzen, John (2008). *Real World Haskell* (<http://book.realworldhaskell.org/read/>). O'Reilly.
- Pratt, Terrence, W. and Marvin V. Zelkowitz. *Programming Languages: Design and Implementation*. 3rd ed. Englewood Cliffs, New Jersey: Prentice Hall, 1996.
- Salus, Peter H. *Functional and Logic Programming Languages*. Vol. 4 of Handbook of Programming Languages. Indianapolis, Indiana: Macmillan Technical Publishing, 1998.
- Thompson, Simon. *Haskell: The Craft of Functional Programming*. Harlow, England: Addison-Wesley Longman Limited, 1996.

External links

- Functional Programming for the Rest of Us (<http://www.defmacro.org/ramblings/fp.html>) An introduction by Slava Akhmechet
- *Functional programming in Python* (by David Mertz): part 1 (http://gnosis.cx/publish/programming/charming_python_13.html), part 2 (http://gnosis.cx/publish/programming/charming_python_16.html), part 3 (http://gnosis.cx/publish/programming/charming_python_19.html)

Lisp (programming language)

Lisp

Paradigm(s)	Multi-paradigm: functional, procedural, reflective, meta
Appeared in	1958
Designed by	John McCarthy
Developer	Steve Russell, Timothy P. Hart, and Mike Levin
Typing discipline	Dynamic, strong
Dialects	Arc, AutoLISP, Clojure, Common Lisp, Emacs Lisp, ISLISP, Newlisp, Picolisp, Racket, Scheme, SKILL, EuLisp, Franz Lisp, Interlisp, LeLisp, Maclisp, MDL, NIL, Portable Standard Lisp, Spice Lisp, T, XLISP, Zetalisp
Influenced by	IPL
Influenced	CLU, Dylan, Falcon, Forth, Haskell, Io, Ioke, JavaScript, Logo, Lua, Mathematica, MDL, ML, Nu, OPS5, Perl, Python, Qi, Shen, Rebol, Racket, Ruby, Smalltalk, Tcl

Lisp (historically, **LISP**) is a family of computer programming languages with a long history and a distinctive, fully parenthesized Polish prefix notation.^[1] Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today; only Fortran is older (by one year). Like Fortran, Lisp has changed a great deal since its early days, and a number of dialects have existed over its history. Today, the most widely known general-purpose Lisp dialects are Common Lisp and Scheme.

Lisp was originally created as a practical mathematical notation for computer programs, influenced by the notation of Alonzo Church's lambda calculus. It quickly became the favored programming language for artificial intelligence (AI) research. As one of the earliest programming languages, Lisp pioneered many ideas in computer science, including tree data structures, automatic storage management, dynamic typing, and the self-hosting compiler.

The name *LISP* derives from "LISt Processing". Linked lists are one of Lisp languages' major data structures, and Lisp source code is itself made up of lists. As a result, Lisp programs can manipulate source code as a data structure, giving rise to the macro systems that allow programmers to create new syntax or even new domain-specific languages embedded in Lisp.

The interchangeability of code and data also gives Lisp its instantly recognizable syntax. All program code is written as *s-expressions*, or parenthesized lists. A function call or syntactic form is written as a list with the function or operator's name first, and the arguments following; for instance, a function f that takes three arguments might be called using

```
(f arg1 arg2 arg3)
```

.

History

Lisp was invented by John McCarthy in 1958 while he was at the Massachusetts Institute of Technology (MIT). McCarthy published its design in a paper in *Communications of the ACM* in 1960, entitled "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I"^[2] ("Part II" was never published). He showed that with a few simple operators and a notation for functions, one can build a Turing-complete language for algorithms.

Information Processing Language was the first AI language, from 1955 or 1956, and already included many of the concepts, such as list-processing and recursion, which came to be used in Lisp.

McCarthy's original notation used bracketed "M-expressions" that would be translated into S-expressions. As an example, the M-expression



A Lisp machine in the MIT Museum

[UNIQ-nowiki-2-9b19ce175d74c32c-QINU](#)

is equivalent to the S-expression

```
(car (cons A B))
```

. Once Lisp was implemented, programmers rapidly chose to use S-expressions, and M-expressions were abandoned. M-expressions surfaced again with short-lived attempts of MLISP^[3] by Horace Enea and CGOL by Vaughan Pratt.

Lisp was first implemented by Steve Russell on an IBM 704 computer. Russell had read McCarthy's paper, and realized (to McCarthy's surprise) that the Lisp *eval* function could be implemented in machine code.^[4] The result was a working Lisp interpreter which could be used to run Lisp programs, or more properly, 'evaluate Lisp expressions.'

Two assembly language macros for the IBM 704 became the primitive operations for decomposing lists: car (Contents of the Address part of Register number) and cdr (Contents of the Decrement part of Register number).^[5] From the context, it is clear that the term "Register" is used here to mean "Memory Register", nowadays called "Memory Location". Lisp dialects still use

`car`

and

`cdr`

(☞ /'kɑr/ and /'kʌdər/) for the operations that return the first item in a list and the rest of the list respectively.

The first complete Lisp compiler, written in Lisp, was implemented in 1962 by Tim Hart and Mike Levin at MIT.^[6] This compiler introduced the Lisp model of incremental compilation, in which compiled and interpreted functions can intermix freely. The language used in Hart and Levin's memo is much closer to modern Lisp style than McCarthy's earlier code.

Lisp was a difficult system to implement with the compiler techniques and stock hardware of the 1970s. Garbage collection routines, developed by then-MIT graduate student Daniel Edwards, made it practical to run Lisp on general-purpose computing systems, but efficiency was still a problem. This led to the creation of Lisp machines:

dedicated hardware for running Lisp environments and programs. Advances in both computer hardware and compiler technology soon made Lisp machines obsolete.

During the 1980s and 1990s, a great effort was made to unify the work on new Lisp dialects (mostly successors to Maclisp like ZetaLisp and NIL (New Implementation of Lisp)) into a single language. The new language, Common Lisp, was somewhat compatible with the dialects it replaced (the book Common Lisp the Language notes the compatibility of various constructs). In 1994, ANSI published the Common Lisp standard, "ANSI X3.226-1994 Information Technology Programming Language Common Lisp."



Steve Russell

Timeline of Lisp dialects^(edit [7])

	1955	1960	1965	1970	1975	1980	1985	1986	1990	1995	2000	2005	2012
Lisp 1.5		Lisp 1.5											
Maclisp				Maclisp									
ZetaLisp					ZetaLisp								
NIL					NIL								
Interlisp				Interlisp									
Common Lisp							Common Lisp						
Scheme						Scheme							
ISLISP									ISLISP				

Connection to artificial intelligence

Since its inception, Lisp was closely connected with the artificial intelligence research community, especially on PDP-10^[8] systems. Lisp was used as the implementation of the programming language Micro Planner which was used in the famous AI system SHRDLU. In the 1970s, as AI research spawned commercial offshoots, the performance of existing Lisp systems became a growing issue.

Genealogy and variants

Over its fifty-year history, Lisp has spawned many variations on the core theme of an S-expression language. Moreover, each given dialect may have several implementations—for instance, there are more than a dozen implementations of Common Lisp.

Differences between dialects may be quite visible—for instance, Common Lisp and Scheme use different keywords to define functions. Within a dialect that is standardized, however, conforming implementations support the same core language, but with different extensions and libraries.

Historically significant dialects

- LISP 1^[9] – First implementation.
- LISP 1.5^[10] – First widely distributed version, developed by McCarthy and others at MIT. So named because it contained several improvements on the original "LISP 1" interpreter, but was not a major restructuring as the planned LISP 2 would be.
- Stanford LISP 1.6^[11] – This was a successor to LISP 1.5 developed at the Stanford AI Lab, and widely distributed to PDP-10 systems running the TOPS-10 operating system. It was rendered obsolete by Maclisp and InterLisp.



John McCarthy

- MACLISP^[12] – developed for MIT's Project MAC (no relation to Apple's Macintosh, nor to McCarthy), direct descendant of LISP 1.5. It ran on the PDP-10 and Multics systems. (MACLISP would later come to be called Maclisp, and is often referred to as MacLisp.)
- InterLisp^[13] – developed at BBN Technologies for PDP-10 systems running the Tenex operating system, later adopted as a "West coast" Lisp for the Xerox Lisp machines as InterLisp-D. A small version called "InterLISP 65" was published for Atari's 6502-based computer line. For quite some time Maclisp and InterLisp were strong competitors.
- Franz Lisp – originally a Berkeley project; later developed by Franz Inc. The name is a humorous deformation of the name "Franz Liszt", and does not refer to Allegro Common Lisp, the dialect of Common Lisp sold by Franz Inc., in more recent years.
- XLISP, which AutoLISP was based on.
- Standard Lisp and Portable Standard Lisp were widely used and ported, especially with the Computer Algebra System REDUCE.
- ZetaLisp, also known as Lisp Machine Lisp – used on the Lisp machines, direct descendant of Maclisp. ZetaLisp had big influence on Common Lisp.
- LeLisp is a French Lisp dialect. One of the first Interface Builders was written in LeLisp.
- Common Lisp (1984), as described by *Common Lisp the Language* – a consolidation of several divergent attempts (ZetaLisp, Spice Lisp, NIL, and S-1 Lisp) to create successor dialects^[14] to Maclisp, with substantive influences from the Scheme dialect as well. This version of Common Lisp was available for wide-ranging platforms and was accepted by many as a de facto standard^[15] until the publication of ANSI Common Lisp (ANSI X3.226-1994).
- Dylan was in its first version a mix of Scheme with the Common Lisp Object System.
- EuLisp – attempt to develop a new efficient and cleaned-up Lisp.
- ISLISP – attempt to develop a new efficient and cleaned-up Lisp. Standardized as ISO/IEC 13816:1997^[16] and later revised as ISO/IEC 13816:2007:^[17] *Information technology – Programming languages, their environments and system software interfaces – Programming language ISLISP*.
- IEEE Scheme – IEEE standard, 1178–1990 (R1995)
- ANSI Common Lisp – an American National Standards Institute (ANSI) standard for Common Lisp, created by subcommittee X3J13, chartered^[18] to begin with *Common Lisp: The Language* as a base document and to work through a public consensus process to find solutions to shared issues of portability of programs and compatibility of Common Lisp implementations. Although formally an ANSI standard, the implementation, sale, use, and influence of ANSI Common Lisp has been and continues to be seen worldwide.
- ACL2 or "A Computational Logic for Applicative Common Lisp", an applicative (side-effect free) variant of Common LISP. ACL2 is both a programming language in which you can model computer systems and a tool to help proving properties of those models.

Since 2000

After having declined somewhat in the 1990s, Lisp has recently experienced a resurgence of interest. Most new activity is focused around open source implementations of Common Lisp, and includes the development of new portable libraries and applications. A new print edition of *Practical Common Lisp* by Peter Seibel, a tutorial for new Lisp programmers, was published in 2005.^[19] It is available free online.

Many new Lisp programmers were inspired by writers such as Paul Graham and Eric S. Raymond to pursue a language others considered antiquated. New Lisp programmers often describe the language as an eye-opening experience and claim to be substantially more productive than in other languages.^[20] This increase in awareness may be contrasted to the "AI winter" and Lisp's brief gain in the mid-1990s.^[21]

Dan Weinreb lists in his survey of Common Lisp implementations^[22] eleven actively maintained Common Lisp implementations. Scieneer Common Lisp is a new commercial implementation forked from CMUCL with a first release in 2002.

The open source community has created new supporting infrastructure: CLiki is a wiki that collects Common Lisp related information, the Common Lisp directory lists resources, #lisp is a popular IRC channel (with support by a Lisp-written Bot), lisppaste supports the sharing and commenting of code snippets, Planet Lisp collects the contents of various Lisp-related blogs, on LispForum users discuss Lisp topics, Lispjobs is a service for announcing job offers and there is a weekly news service, *Weekly Lisp News*. *Common-lisp.net* is a hosting site for open source Common Lisp projects.

50 years of Lisp (1958–2008) has been celebrated at LISP50@OOPSLA.^[23] There are regular local user meetings in Boston, Vancouver, and Hamburg. Other events include the European Common Lisp Meeting, the European Lisp Symposium and an International Lisp Conference.

The Scheme community actively maintains over twenty implementations. Several significant new implementations (Chicken, Gambit, Gauche, Ikarus, Larceny, Ypsilon) have been developed in the last few years. The Revised⁵ Report on the Algorithmic Language Scheme^[24] standard of Scheme was widely accepted in the Scheme community. The Scheme Requests for Implementation process has created a lot of quasi standard libraries and extensions for Scheme. User communities of individual Scheme implementations continue to grow. A new language standardization process was started in 2003 and led to the R⁶RS Scheme standard in 2007. Academic use of Scheme for teaching computer science seems to have declined somewhat. Some universities are no longer using Scheme in their computer science introductory courses.

There are several new dialects of Lisp: Arc, Nu, and Clojure.

Major dialects

The two major dialects of Lisp used for general-purpose programming today are Common Lisp and Scheme. These languages represent significantly different design choices.

Common Lisp is a successor to MacLisp. The primary influences were Lisp Machine Lisp, MacLisp, NIL, S-1 Lisp, Spice Lisp, and Scheme.^[25] It has many of the features of Lisp Machine Lisp (a large Lisp dialect used to program Lisp Machines), but was designed to be efficiently implementable on any personal computer or workstation. Common Lisp has a large language standard including many built-in data types, functions, macros and other language elements, as well as an object system (Common Lisp Object System or shorter CLOS). Common Lisp also borrowed certain features from Scheme such as lexical scoping and lexical closures.

Scheme (designed earlier) is a more minimalist design, with a much smaller set of standard features but with certain implementation features (such as tail-call optimization and full continuations) not necessarily found in Common Lisp.

Scheme is a statically scoped and properly tail-recursive dialect of the Lisp programming language invented by Guy Lewis Steele Jr. and Gerald Jay Sussman. It was designed to have exceptionally clear and simple semantics and few

different ways to form expressions. A wide variety of programming paradigms, including imperative, functional, and message passing styles, find convenient expression in Scheme. Scheme continues to evolve with a series of standards (Revisedⁿ Report on the Algorithmic Language Scheme) and a series of Scheme Requests for Implementation.

Clojure is a recent dialect of Lisp that principally targets the Java Virtual Machine, as well as the CLR, the Python VM, and compiling to JavaScript. It is designed to be a pragmatic general-purpose language. Clojure draws considerable influences from Haskell and places a very strong emphasis on immutability.^[26] Clojure is a compiled language, as it compiles directly to JVM bytecode, yet remains completely dynamic. Every feature supported by Clojure is supported at runtime. Clojure provides access to Java frameworks and libraries, with optional type hints and type inference, so that calls to Java can avoid reflection and enable fast primitive operations.

In addition, Lisp dialects are used as scripting languages in a number of applications, with the most well-known being Emacs Lisp in the Emacs editor, AutoLisp and later Visual Lisp in AutoCAD, Nyquist in Audacity. The small size of a minimal but useful Scheme interpreter makes it particularly popular for embedded scripting. Examples include SIOD and TinyScheme, both of which have been successfully embedded in the GIMP image processor under the generic name "Script-fu".^[27] LIBREP, a Lisp interpreter by John Harper originally based on the Emacs Lisp language, has been embedded in the Sawfish window manager.^[28] The Guile interpreter is used in GnuCash. Within GCC, the MELT plugin provides a Lisp-y dialect, translated into C, to extend the compiler by coding additional passes (in MELT).

Language innovations

Lisp was the first homoiconic programming language: the primary representation of program code is the same type of list structure that is also used for the main data structures. As a result, Lisp functions can be manipulated, altered or even created within a Lisp program without extensive parsing or manipulation of binary machine code. This is generally considered one of the primary advantages of the language with regard to its expressive power, and makes the language amenable to metacircular evaluation.

The ubiquitous *if-then-else* structure, now taken for granted as an essential element of any programming language, was invented by McCarthy for use in Lisp, where it saw its first appearance in a more general form (the cond structure). It was inherited by ALGOL, which popularized it.

Lisp deeply influenced Alan Kay, the leader of the research on Smalltalk, and then in turn Lisp was influenced by Smalltalk, by adopting object-oriented programming features (classes, instances, etc.) in the late 1970s. The Flavours object system (later CLOS) introduced multiple inheritance.

Lisp introduced the concept of automatic garbage collection, in which the system walks the heap looking for unused memory. Most of the modern sophisticated garbage collection algorithms such as generational garbage collection were developed for Lisp.

Largely because of its resource requirements with respect to early computing hardware (including early microprocessors), Lisp did not become as popular outside of the AI community as Fortran and the ALGOL-descended C language. Newer languages such as Java and Python have incorporated some limited versions of some of the features of Lisp, but are necessarily unable to bring the coherence and synergy of the full concepts found in Lisp. Because of its suitability to complex and dynamic applications, Lisp is currently enjoying some resurgence of popular interest.

Syntax and semantics

Note: This article's examples are written in Common Lisp (though most are also valid in Scheme).

Symbolic expressions

Lisp is an expression-oriented language. Unlike most other languages, no distinction is made between "expressions" and "statements"; all code and data are written as expressions. When an expression is *evaluated*, it produces a value (in Common Lisp, possibly multiple values), which then can be embedded into other expressions. Each value can be any data type.

McCarthy's 1958 paper introduced two types of syntax: S-expressions (Symbolic expressions, also called "sexps"), which mirror the internal representation of code and data; and M-expressions (Meta Expressions), which express functions of S-expressions. M-expressions never found favor, and almost all Lisps today use S-expressions to manipulate both code and data.

The use of parentheses is Lisp's most immediately obvious difference from other programming language families. As a result, students have long given Lisp nicknames such as *Lost In Stupid Parentheses*, or *Lots of Irritating Superfluous Parentheses*.^[29] However, the S-expression syntax is also responsible for much of Lisp's power: the syntax is extremely regular, which facilitates manipulation by computer. However, the syntax of Lisp is not limited to traditional parentheses notation. It can be extended to include alternative notations. XMLisp, for instance, is a Common Lisp extension that employs the metaobject-protocol to integrate S-expressions with the Extensible Markup Language (XML).

The reliance on expressions gives the language great flexibility. Because Lisp functions are themselves written as lists, they can be processed exactly like data. This allows easy writing of programs which manipulate other programs (metaprogramming). Many Lisp dialects exploit this feature using macro systems, which enables extension of the language almost without limit.

Lists

A Lisp list is written with its elements separated by whitespace, and surrounded by parentheses. For example,

```
(1 2 foo)
```

is a list whose elements are three *atoms*: the values

```
1
```

```
,
```

```
2
```

, and foo. These values are implicitly typed: they are respectively two integers and a Lisp-specific data type called a "symbol", and do not have to be declared as such.

The empty list

```
()
```

is also represented as the special atom

```
nil
```

. This is the only entity in Lisp which is both an atom and a list.

Expressions are written as lists, using prefix notation. The first element in the list is the name of a *form*, i.e., a function, operator, macro, or "special operator" (see below.) The remainder of the list are the arguments. For example, the function

```
list
```

returns its arguments as a list, so the expression

```
(list '1 '2 'foo)
```

evaluates to the list

```
(1 2 foo)
```

. The "quote" before the arguments in the preceding example is a "special operator" which prevents the quoted arguments from being evaluated (not strictly necessary for the numbers, since 1 evaluates to 1, etc.). Any unquoted expressions are recursively evaluated before the enclosing expression is evaluated. For example,

```
(list 1 2 (list 3 4))
```

evaluates to the list

```
(1 2 (3 4))
```

. Note that the third argument is a list; lists can be nested.

Operators

Arithmetic operators are treated similarly. The expression

```
(+ 1 2 3 4)
```

evaluates to 10. The equivalent under infix notation would be "

```
1 + 2 + 3 + 4
```

". Arithmetic operators in Lisp are variadic (or *n-ary*), able to take any number of arguments.

"Special operators" (sometimes called "special forms") provide Lisp's control structure. For example, the special operator

```
if
```

takes three arguments. If the first argument is non-nil, it evaluates to the second argument; otherwise, it evaluates to the third argument. Thus, the expression

```
(if nil  
    (list 1 2 "foo")  
    (list 3 4 "bar"))
```

evaluates to

```
(3 4 "bar")
```

. Of course, this would be more useful if a non-trivial expression had been substituted in place of

```
nil
```

.

Lambda expressions

Another special operator,

```
lambda
```

, is used to bind variables to values which are then evaluated within an expression. This operator is also used to create functions: the arguments to

```
lambda
```

are a list of arguments, and the expression or expressions to which the function evaluates (the returned value is the value of the last expression that is evaluated). The expression

```
(lambda (arg) (+ arg 1))
```

evaluates to a function that, when applied, takes one argument, binds it to

```
arg
```

and returns the number one greater than that argument. Lambda expressions are treated no differently from named functions; they are invoked the same way. Therefore, the expression

```
((lambda (arg) (+ arg 1)) 5)
```

evaluates to

```
6
```

.

Atoms

In the original **LISP** there were two fundamental data types: atoms and lists. A list was a finite ordered sequence of elements, where each element is in itself either an atom or a list, and an atom was a number or a symbol. A symbol was essentially a unique named item, written as an Alphanumeric string in source code, and used either as a variable name or as a data item in symbolic processing. For example, the list

```
(FOO (BAR 1) 2)
```

contains three elements: the symbol FOO, the list

```
(BAR 1)
```

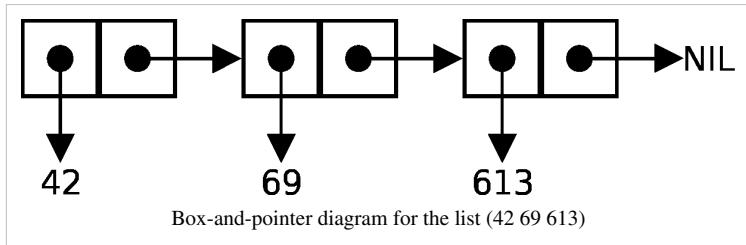
, and the number 2.

The essential difference between atoms and lists was that atoms were immutable and unique. Two atoms that appeared in different places in source code but were written in exactly the same way represented the same object, whereas each list was a separate object that could be altered independently of other lists and could be distinguished from other lists by comparison operators.

As more data types were introduced in later Lisp dialects, and programming styles evolved, the concept of an atom lost importance. Many dialects still retained the predicate *atom* for legacy compatibility, defining it true for any object which is not a cons.

Conses and lists

A Lisp list is a singly linked list. Each cell of this list is called a *cons* (in Scheme, a *pair*), and is composed of two pointers, called the *car* and *cdr*. These are equivalent to the



`data`

and

`next`

fields discussed in the article *linked list*, respectively.

Of the many data structures that can be built out of cons cells, one of the most basic is called a *proper list*. A proper list is either the special

`nil`

(empty list) symbol, or a cons in which the

`car`

points to a datum (which may be another cons structure, such as a list), and the

`cdr`

points to another proper list.

If a given cons is taken to be the head of a linked list, then its car points to the first element of the list, and its cdr points to the rest of the list. For this reason, the

`car`

and

`cdr`

functions are also called

`first`

and

`rest`

when referring to conses which are part of a linked list (rather than, say, a tree).

Thus, a Lisp list is not an atomic object, as an instance of a container class in C++ or Java would be. A list is nothing more than an aggregate of linked conses. A variable which refers to a given list is simply a pointer to the first cons in the list. Traversal of a list can be done by "cdring down" the list; that is, taking successive cdrs to visit each cons of the list; or by using any of a number of higher-order functions to map a function over a list.

Because conses and lists are so universal in Lisp systems, it is a common misconception that they are Lisp's only data structures. In fact, all but the most simplistic Lisps have other data structures – such as vectors (arrays), hash

tables, structures, and so forth.

S-expressions represent lists

Parenthesized S-expressions represent linked list structures. There are several ways to represent the same list as an S-expression. A cons can be written in *dotted-pair notation* as

```
(a . b)
```

, where

```
a
```

is the car and

```
b
```

the cdr. A longer proper list might be written

```
(a . (b . (c . (d . nil)))))
```

in dotted-pair notation. This is conventionally abbreviated as

```
(a b c d)
```

in *list notation*. An improper list^[30] may be written in a combination of the two – as

```
(a b c . d)
```

for the list of three conses whose last cdr is

```
d
```

(i.e., the list

```
(a . (b . (c . d)))
```

in fully specified form).

List-processing procedures

Lisp provides many built-in procedures for accessing and controlling lists. Lists can be created directly with the

```
list
```

procedure, which takes any number of arguments, and returns the list of these arguments.

```
(list 1 2 'a 3)  
;Output: (1 2 a 3)
```

```
(list 1 '(2 3) 4)  
;Output: (1 (2 3) 4)
```

Because of the way that lists are constructed from cons pairs, the

```
[ [cons] ]
```

procedure can be used to add an element to the front of a list. Note that the

```
cons
```

procedure is asymmetric in how it handles list arguments, because of how lists are constructed.

```
(cons 1 '(2 3))
;Output: (1 2 3)

(cons '(1 2) '(3 4))
;Output: ((1 2) 3 4)
```

The

[[append]]

procedure appends two (or more) lists to one another. Because Lisp lists are linked lists, appending two lists has asymptotic time complexity $O(n)$

```
(append '(1 2) '(3 4))
;Output: (1 2 3 4)

(append '(1 2 3) '() '(a) '(5 6))
;Output: (1 2 3 a 5 6)
```

Shared structure

Lisp lists, being simple linked lists, can share structure with one another. That is to say, two lists can have the same *tail*, or final sequence of conses. For instance, after the execution of the following Common Lisp code:

```
(setf foo (list 'a 'b 'c))
(setf bar (cons 'x (cdr foo)))
```

the lists

foo

and

bar

are

(a b c)

and

(x b c)

respectively. However, the tail

(b c)

is the same structure in both lists. It is not a copy; the cons cells pointing to

b

and

c

are in the same memory locations for both lists.

Sharing structure rather than copying can give a dramatic performance improvement. However, this technique can interact in undesired ways with functions that alter lists passed to them as arguments. Altering one list, such as by replacing the

c

with a

goose

, will affect the other:

```
(setf (third foo) 'goose)
```

This changes

foo

to

(a b goose)

, but thereby also changes

bar

to

(x b goose)

– a possibly unexpected result. This can be a source of bugs, and functions which alter their arguments are documented as *destructive* for this very reason.

Aficionados of functional programming avoid destructive functions. In the Scheme dialect, which favors the functional style, the names of destructive functions are marked with a cautionary exclamation point, or "bang"—such as

set-car!

(read *set car bang*), which replaces the car of a cons. In the Common Lisp dialect, destructive functions are commonplace; the equivalent of

set-car!

is named

rplaca

for "replace car." This function is rarely seen however as Common Lisp includes a special facility,

setf

, to make it easier to define and use destructive functions. A frequent style in Common Lisp is to write code functionally (without destructive calls) when prototyping, then to add destructive calls as an optimization where it is safe to do so.

Self-evaluating forms and quoting

Lisp evaluates expressions which are entered by the user. Symbols and lists evaluate to some other (usually, simpler) expression – for instance, a symbol evaluates to the value of the variable it names;

```
(+ 2 3)
```

evaluates to

```
5
```

. However, most other forms evaluate to themselves: if you enter

```
5
```

into Lisp, it returns

```
5
```

. Any expression can also be marked to prevent it from being evaluated (as is necessary for symbols and lists). This is the role of the

quote

special operator, or its abbreviation

```
'
```

(a single quotation mark). For instance, usually if you enter the symbol

```
foo
```

you will get back the value of the corresponding variable (or an error, if there is no such variable). If you wish to refer to the literal symbol, you enter

```
(quote foo)
```

or, usually,

```
' foo
```

. Both Common Lisp and Scheme also support the *backquote* operator (known as *quasiquote* in Scheme), entered with the

```
`
```

character. This is almost the same as the plain quote, except it allows expressions to be evaluated and their values interpolated into a quoted list with the comma and comma-at operators. If the variable

```
snue
```

has the value

```
(bar baz)
```

then

```
` (foo , snue)
```

evaluates to

```
(foo (bar baz))
```

, while

```
`(foo ,@snue)
```

evaluates to

```
(foo bar baz)
```

. The backquote is most frequently used in defining macro expansions.

Self-evaluating forms and quoted forms are Lisp's equivalent of literals. It may be possible to modify the values of (mutable) literals in program code. For instance, if a function returns a quoted form, and the code that calls the function modifies the form, this may alter the behavior of the function on subsequent iterations.

```
(defun should-be-constant ()
  '(one two three))

(let ((stuff (should-be-constant)))
  (setf (third stuff) 'bizarre)) ; bad!

(should-be-constant) ; returns (one two bizarre)
```

Modifying a quoted form like this is generally considered bad style, and is defined by ANSI Common Lisp as erroneous (resulting in "undefined" behavior in compiled files, because the file-compiler can coalesce similar constants, put them in write-protected memory, etc.).

Lisp's formalization of quotation has been noted by Douglas Hofstadter (in *Gödel, Escher, Bach*) and others as an example of the philosophical idea of self-reference.

Scope and closure

The modern Lisp family splits over the use of dynamic or static (aka lexical) scope. Clojure, Common Lisp and Scheme make use of static scoping by default, while Newlisp, Picolisp and the embedded languages in Emacs and AutoCAD use dynamic scoping.

List structure of program code; exploitation by macros and compilers

A fundamental distinction between Lisp and other languages is that in Lisp, the textual representation of a program is simply a human-readable description of the same internal data structures (linked lists, symbols, number, characters, etc.) as would be used by the underlying Lisp system.

Lisp uses this to implement a very powerful macro system. Like other macro languages such as C, a macro returns code that can then be compiled. However, unlike C macros, the macros are Lisp functions and so can exploit the full power of Lisp.

Further, because Lisp code has the same structure as lists, macros can be built with any of the list-processing functions in the language. In short, anything that Lisp can do to a data structure, Lisp macros can do to code. In contrast, in most other languages, the parser's output is purely internal to the language implementation and cannot be manipulated by the programmer.

This makes it easy to develop *efficient* languages within languages. For example, the Common Lisp Object System can be implemented cleanly as a language extension using macros. This means that if an application requires a different inheritance mechanism, it can use a different object system. This is in stark contrast to most other

languages, for example Java does not support multiple inheritance and there is no reasonable way to add it.

In simplistic Lisp implementations, this list structure is directly interpreted to run the program; a function is literally a piece of list structure which is traversed by the interpreter in executing it. However, most substantial Lisp systems also include a compiler. The compiler translates list structure into machine code or bytecode for execution. This code can run as fast as code compiled in conventional languages such as C.

Macros expand before the compilation step, and thus offer some interesting options. If a program needs a precomputed table, then a macro might create the table at compile time, so the compiler need only output the table and need not call code to create the table at run time. Some Lisp implementations even have a mechanism, `eval-when`, that allows code to be present during compile time (when a macro would need it), but not present in the emitted module.^[31]

Evaluation and the read–eval–print loop

Lisp languages are frequently used with an interactive command line, which may be combined with an integrated development environment. The user types in expressions at the command line, or directs the IDE to transmit them to the Lisp system. Lisp *reads* the entered expressions, *evaluates* them, and *prints* the result. For this reason, the Lisp command line is called a "read–eval–print loop", or *REPL*.

The basic operation of the REPL is as follows. This is a simplistic description which omits many elements of a real Lisp, such as quoting and macros.

The

```
read
```

function accepts textual S-expressions as input, and parses them into an internal data structure. For instance, if you type the text

```
(+ 1 2)
```

at the prompt,

```
read
```

translates this into a linked list with three elements: the symbol

```
+
```

, the number 1, and the number 2. It so happens that this list is also a valid piece of Lisp code; that is, it can be evaluated. This is because the car of the list names a function—the addition operation.

Note that a

```
foo
```

will be read as a single symbol.

```
123
```

will be read as the number 123.

```
"123"
```

will be read as the string "123".

The

```
eval
```

function evaluates the data, returning zero or more other Lisp data as a result. Evaluation does not have to mean interpretation; some Lisp systems compile every expression to native machine code. It is simple, however, to describe evaluation as interpretation: To evaluate a list whose car names a function,

```
eval
```

first evaluates each of the arguments given in its cdr, then applies the function to the arguments. In this case, the function is addition, and applying it to the argument list

```
(1 2)
```

yields the answer

```
3
```

. This is the result of the evaluation.

The symbol

```
foo
```

evaluates to the value of the symbol foo. Data like the string "123" evaluates to the same string. The list

```
(quote (1 2 3))
```

evaluates to the list (1 2 3).

It is the job of the

```
print
```

function to represent output to the user. For a simple result such as

```
3
```

this is trivial. An expression which evaluated to a piece of list structure would require that

```
print
```

traverse the list and print it out as an S-expression.

To implement a Lisp REPL, it is necessary only to implement these three functions and an infinite-loop function. (Naturally, the implementation of

```
eval
```

will be complicated, since it must also implement all special operators like

```
if
```

or

```
lambda
```

.) This done, a basic REPL itself is but a single line of code:

```
(loop (print (eval (read))))
```

The Lisp REPL typically also provides input editing, an input history, error handling and an interface to the debugger.

Lisp is usually evaluated eagerly. In Common Lisp, arguments are evaluated in applicative order ('leftmost innermost'), while in Scheme order of arguments is undefined, leaving room for optimization by a compiler.

Control structures

Lisp originally had very few control structures, but many more were added during the language's evolution. (Lisp's original conditional operator,

`cond`

, is the precursor to later

`if-then-else`

structures.)

Programmers in the Scheme dialect often express loops using tail recursion. Scheme's commonality in academic computer science has led some students to believe that tail recursion is the only, or the most common, way to write iterations in Lisp, but this is incorrect. All frequently seen Lisp dialects have imperative-style iteration constructs, from Scheme's

`do`

loop to Common Lisp's complex

`loop`

expressions. Moreover, the key issue that makes this an objective rather than subjective matter is that Scheme makes specific requirements for the handling of tail calls, and consequently the reason that the use of tail recursion is generally encouraged for Scheme is that the practice is expressly supported by the language definition itself. By contrast, ANSI Common Lisp does not require^[32] the optimization commonly referred to as tail call elimination. Consequently, the fact that tail recursive style as a casual replacement for the use of more traditional iteration constructs (such as

`do`

,

`dolist`

or

`loop`

) is discouraged^[33] in Common Lisp is not just a matter of stylistic preference, but potentially one of efficiency (since an apparent tail call in Common Lisp may not compile as a simple jump) and program correctness (since tail recursion may increase stack use in Common Lisp, risking stack overflow).

Some Lisp control structures are *special operators*, equivalent to other languages' syntactic keywords. Expressions using these operators have the same surface appearance as function calls, but differ in that the arguments are not necessarily evaluated—or, in the case of an iteration expression, may be evaluated more than once.

In contrast to most other major programming languages, Lisp allows the programmer to implement control structures using the language itself. Several control structures are implemented as Lisp macros, and can even be macro-expanded by the programmer who wants to know how they work.

Both Common Lisp and Scheme have operators for non-local control flow. The differences in these operators are some of the deepest differences between the two dialects. Scheme supports *re-entrant continuations* using the

`call/cc`

procedure, which allows a program to save (and later restore) a particular place in execution. Common Lisp does not support re-entrant continuations, but does support several ways of handling escape continuations.

Frequently, the same algorithm can be expressed in Lisp in either an imperative or a functional style. As noted above, Scheme tends to favor the functional style, using tail recursion and continuations to express control flow. However, imperative style is still quite possible. The style preferred by many Common Lisp programmers may seem more familiar to programmers used to structured languages such as C, while that preferred by Schemers more closely resembles pure-functional languages such as Haskell.

Because of Lisp's early heritage in list processing, it has a wide array of higher-order functions relating to iteration over sequences. In many cases where an explicit loop would be needed in other languages (like a

`for`

loop in C) in Lisp the same task can be accomplished with a higher-order function. (The same is true of many functional programming languages.)

A good example is a function which in Scheme is called

`map`

and in Common Lisp is called

`mapcar`

. Given a function and one or more lists,

`mapcar`

applies the function successively to the lists' elements in order, collecting the results in a new list:

```
(mapcar #'+ '(1 2 3 4 5) '(10 20 30 40 50))
```

This applies the

`+`

function to each corresponding pair of list elements, yielding the result

```
(11 22 33 44 55)
```

Examples

Here are examples of Common Lisp code.

The basic "Hello world" program:

```
(print "Hello world")
```

Lisp syntax lends itself naturally to recursion. Mathematical problems such as the enumeration of recursively defined sets are simple to express in this notation.

Evaluate a number's factorial:

```
(defun factorial (n)
  (if (<= n 1)
      1
```

```
(* n (factorial (- n 1)))))
```

An alternative implementation, often faster than the previous version if the Lisp system has tail recursion optimization:

```
(defun factorial (n &optional (acc 1))
  (if (<= n 1)
      acc
      (factorial (- n 1) (* acc n))))
```

Contrast with an iterative version which uses Common Lisp's

```
loop
```

macro:

```
(defun factorial (n)
  (loop for i from 1 to n
        for fac = 1 then (* fac i)
        finally (return fac)))
```

The following function reverses a list. (Lisp's built-in *reverse* function does the same thing.)

```
(defun -reverse (list)
  (let ((return-value '()))
    (dolist (e list) (push e return-value))
    return-value))
```

Object systems

Various object systems and models have been built on top of, alongside, or into Lisp, including:

- The Common Lisp Object System, CLOS, is an integral part of ANSI Common Lisp. CLOS descended from New Flavors and CommonLOOPS. ANSI Common Lisp was the first standardized object-oriented programming language (1994, ANSI X3J13).
- ObjectLisp^[34] or Object Lisp, used by Lisp Machines Incorporated and early versions of Macintosh Common Lisp
- LOOPS (Lisp Object-Oriented Programming System) and the later CommonLOOPS
- Flavors, built at MIT, and its descendant New Flavors (developed by Symbolics).
- KR (short for Knowledge Representation), a constraints-based object system developed to aid the writing of Garnet, a GUI library for Common Lisp.
- KEE used an object system called UNITS and integrated it with an inference engine^[35] and a truth maintenance system (ATMS).

References

- [1] Edwin D. Reilly (2003). *Milestones in computer science and information technology* (<http://books.google.com/books?id=JTYPKxug49IC&pg=PA157>). Greenwood Publishing Group. pp. 156–157. ISBN 978-1-57356-521-9. .
- [2] John McCarthy. "Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I" (<http://www-formal.stanford.edu/jmc/recursive.html>) . Retrieved 2006-10-13.
- [3] David Canfield Smith. "MLISP Users Manual" (<http://www.softwarepreservation.org/projects/LISP/stanford/Smith-MLISP-AIM-84.pdf>) . Retrieved 2006-10-13.
- [4] According to what reported by Paul Graham in *Hackers & Painters*, p. 185, McCarthy said: "Steve Russell said, look, why don't I program this *eval*..., and I said to him, ho, ho, you're confusing theory with practice, this *eval* is intended for reading, not for computing. But he went ahead and did it. That is, he compiled the *eval* in my paper into IBM 704 machine code, fixing bug, and then advertised this as a Lisp interpreter, which it certainly was. So at that point Lisp had essentially the form that it has today..."
- [5] John McCarthy. "LISP prehistory - Summer 1956 through Summer 1958" (<http://www-formal.stanford.edu/jmc/history/lisp/node2.html>) . Retrieved 2010-03-14.
- [6] Tim Hart and Mike Levin. "AI Memo 39-The new compiler" (<ftp://publications.ai.mit.edu/ai-publications/pdf/AIM-039.pdf>) . Retrieved 2006-10-13.
- [7] <http://en.wikipedia.org/w/index.php?title=Template:Lisp&action=edit>
- [8] The 36-bit word size of the PDP-6/PDP-10 was influenced by the usefulness of having two Lisp 18-bit pointers in a single word. Peter J. Hurley (18 October 1990). "@tut.cis.ohio-state.edu The History of TOPS or Life in the Fast ACs (news:84950)". [lt.folklore.computers](http://groups.google.com/group/alt.folklore.computers/browse_thread/thread/6e5602ce733d0ec/17597705ae289112) (news:a). Web link (http://groups.google.com/group/alt.folklore.computers/browse_thread/thread/6e5602ce733d0ec/17597705ae289112). "The PDP-6 project started in early 1963, as a 24-bit machine. It grew to 36 bits for LISP, a design goal.".
- [9] McCarthy, J.; Brayton, R.; Edwards, D.; Fox, P.; Hodes, L.; Luckham, D.; Maling, K.; Park, D. et al. (March 1960). *LISP I Programmers Manual* (http://history.siam.org/sup/Fox_1960_LISP.pdf). Boston, Massachusetts: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory. Accessed May 11, 2010.
- [10] McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, Timothy P.; Levin, Michael I. (1962; 2nd Edition, 15th printing, 1985). *LISP 1.5 Programmer's Manual* (<http://www.softwarepreservation.org/projects/LISP/book/LISP 1.5 Programmers Manual.pdf>). MIT Press. ISBN 0-262-13011-4. .
- [11] Quam, Lynn H.; Diffler, Whitfield (PDF). *Stanford LISP 1.6 Manual* (<http://www.softwarepreservation.org/projects/LISP/stanford-SAILON-28.6.pdf>) .
- [12] "MacLisp Reference Manual" (<http://web.archive.org/web/20071214064433/http://zane.brouhaha.com/~healyzh/doc/lisp.doc.txt>). March 3, 1979. Archived from the original (<http://zane.brouhaha.com/~healyzh/doc/lisp.doc.txt>) on 2007-12-14. .
- [13] Teitelman, Warren (1974) (PDF). *InterLisp Reference Manual* (http://www.bitsavers.org/pdf/xerox/interlisp/1974_InterlispRefMan.pdf) .
- [14] Steele, Guy L., Jr.. "Purpose" (<http://www.cs.cmu.edu/Groups/AI/html/cltl/clm/node6.html>). *Common Lisp the Language* (2nd ed.). ISBN 0-13-152414-3. .
- [15] Kantrowitz, Mark; Margolin, Barry (20 February 1996). "History: Where did Lisp come from?" (<http://www.cs.cmu.edu/Groups/AI/html/faqs/lang/lisp/part2/faq-doc-13.html>). *FAQ: Lisp Frequently Asked Questions 2/7*. .
- [16] ISO/IEC 13816:1997 (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=22987)
- [17] ISO/IEC 13816:2007 (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=44338)
- [18] "X3J13 Charter" (<http://www.nhplace.com/kent/CL/x3j13-86-020.html>). .
- [19] Siebel, Peter (2005). *Practical Common Lisp* (<http://gigamonkeys.com/book/>). Apress. ISBN 978-1-59059-239-7. .
- [20] "The Road To Lisp Survey" (http://wiki.alu.org/The_Road_To_Lisp_Survey) . Retrieved 2006-10-13.
- [21] Trends for the Future (<http://www.faqs.org/docs/artu/ch14s05.html>)
- [22] Weinreb, Daniel. "Common Lisp Implementations: A Survey" (<http://common-lisp.net/~dlw/LispSurvey.html>) . Retrieved 4 April 2012.
- [23] LISP50@OOPSLA (<http://www.lisp50.org/>)
- [24] <http://www.schemers.org/Documents/Standards/R5RS/>
- [25] Chapter 1.1.2, History, ANSI CL Standard
- [26] An In-Depth Look at Clojure Collections (<http://www.infoq.com/articles/in-depth-look-clojure-collections>), Retrieved 2012-06-24
- [27] Script-fu In GIMP 2.4 (<http://www.gimp.org/docs/script-fu-update.html>), Retrieved 2009-10-29
- [28] librep (<http://sawfish.wikia.com/wiki/Librep>) at Sawfish Wikia, retrieved 2009-10-29
- [29] "The Jargon File - Lisp" (<http://www.catb.org/~esr/jargon/html/L/LISP.html>) . Retrieved 2006-10-13.
- [30] NB: a so-called "dotted list" is only one kind of "improper list". The other kind is the "circular list" where the cons cells form a loop. Typically this is represented using #n=(...) to represent the target cons cell that will have multiple references, and #n# is used to refer to this cons. For instance, (#1=(a b) . #1#) would normally be printed as ((a b) a b) (without circular structure printing enabled), but makes the reuse of the cons cell clear. #1=(a . #1#) cannot normally be printed as it is circular, the CDR of the cons cell defined by #1= is itself.
- [31] http://www.gnu.org/software/emacs/manual/html_node/cl/Time-of-Evaluation.html
- [32] 3.2.2.3 Semantic Constraints (http://www.lispworks.com/documentation/HyperSpec/Body/03_bbc.htm) in *Common Lisp HyperSpec* (<http://www.lispworks.com/documentation/HyperSpec/Front/index.htm>)

- [33] 4.3. Control Abstraction (Recursion vs. Iteration) in Tutorial on Good Lisp Programming Style (<http://www.cs.umd.edu/~nau/cmse421/norvig-lisp-style.pdf>) by Pitman and Norvig, August, 1993.
- [34] pg 17 of Bobrow 1986
- [35] Veitch, p 108, 1988

Further reading

- McCarthy, John (1979-02-12). "The implementation of Lisp" (<http://www-formal.stanford.edu/jmc/history/lisp/node3.html>). *History of Lisp*. Stanford University. Retrieved 2008-10-17.
- Steele, Jr., Guy L.; Richard P. Gabriel (1993). "The evolution of Lisp" (<http://www.dreamsongs.com/NewFiles/HOPL2-Uncut.pdf>). *The second ACM SIGPLAN conference on History of programming languages*. New York, NY: ACM, ISBN 0-89791-570-4. pp. 231–270. ISBN 0-89791-570-4. Retrieved 2008-10-17.
- Veitch, Jim (1998). "A history and description of CLOS". In Salus, Peter H. *Handbook of programming languages. Volume IV, Functional and logic programming languages* (first ed.). Indianapolis, IN: Macmillan Technical Publishing. pp. 107–158. ISBN 1-57870-011-6
- Abelson, Harold; Sussman, Gerald Jay; Sussman, Julie (1996). *Structure and Interpretation of Computer Programs* (2nd ed.). MIT Press. ISBN 0-262-01153-0.
- My Lisp Experiences and the Development of GNU Emacs (<http://www.gnu.org/gnu/rms-lisp.html>), transcript of Richard Stallman's speech, 28 October 2002, at the International Lisp Conference
- Graham, Paul (2004). *Hackers & Painters. Big Ideas from the Computer Age*. O'Reilly. ISBN 0-596-00662-4.
- Berkeley, Edmund C.; Bobrow, Daniel G., eds. (March 1964). *The Programming Language LISP: Its Operation and Applications* (http://www.softwarepreservation.org/projects/LISP/book/III_LispBook_Apr66.pdf). Cambridge, Massachusetts: MIT Press.
- Weissman, Clark (1967). *LISP 1.5 Primer* (http://www.softwarepreservation.org/projects/LISP/book/Weissmann_LISP1.5_Primer_1967.pdf). Belmont, California: Dickenson Publishing Company Inc..

External links

History

- History of Lisp (<http://www-formal.stanford.edu/jmc/history/lisp/lisp.html>) – John McCarthy's history of 12 February 1979
- Lisp History (<http://www8.informatik.uni-erlangen.de/html/lisp-enter.html>) – Herbert Stoyan's history compiled from the documents (acknowledged by McCarthy as more complete than his own, see: McCarthy's history links (<http://www-formal.stanford.edu/jmc/history/>))
- History of LISP at the Computer History Museum (<http://www.softwarepreservation.org/projects/LISP/>)

Associations and meetings

- Association of Lisp Users (<http://www.alu.org/>)
- European Common Lisp Meeting (<http://www.weitz.de/eclm2009/>)
- European Lisp Symposium (<http://european-lisp-symposium.org/>)
- International Lisp Conference (<http://www.international-lisp-conference.org/>)

Books and tutorials

- Casting SPELs in Lisp (<http://www.lisperati.com/casting.html>), a comic-book style introductory tutorial
- On Lisp (<http://paulgraham.com/onlisptext.html>), a free book by Paul Graham

Interviews

- Oral history interview with John McCarthy (<http://purl.umn.edu/107476>) at Charles Babbage Institute, University of Minnesota, Minneapolis. McCarthy discusses his role in the development of time-sharing at the Massachusetts Institute of Technology. He also describes his work in artificial intelligence (AI) funded by the Advanced Research Projects Agency, including logic-based AI (LISP) and robotics.

- Interview (<http://www.se-radio.net/2008/01/episode-84-dick-gabriel-on-lisp/>) with Richard P. Gabriel (Podcast)

Resources

- Common Lisp directory (<http://www.cl-user.net/>)
- Lisp FAQ Index (<http://www.faqs.org/faqs/lisp-faq/>)
- lisppaste (<http://paste.lisp.org/>)
- Planet Lisp (<http://planet.lisp.org/>)
- Weekly Lisp News (<http://lispnews.wordpress.com/>)
- Lisp (<http://www.dmoz.org/Computers/Programming/Languages/Lisp/>) at the Open Directory Project

Pattern matching

In computer science, **pattern matching** is the act of checking a perceived **sequence** of tokens for the presence of the constituents of some pattern. In contrast to pattern recognition, the match usually has to be exact. The patterns generally have the form of either sequences or tree structures. Uses of pattern matching include outputting the locations (if any) of a pattern within a token sequence, to output some component of the matched pattern, and to substitute the matching pattern with some other token sequence (i.e., search and replace).

Sequence patterns (e.g., a text string) are often described using regular expressions and matched using techniques such as backtracking.

Tree patterns are used in some programming languages as a general tool to process data based on its structure, e.g., Haskell, ML and the symbolic mathematics language Mathematica have special syntax for expressing tree patterns and a language construct for conditional execution and value retrieval based on it. For simplicity and efficiency reasons, these tree patterns lack some features that are available in regular expressions.

Often it is possible to give alternative patterns that are tried one by one, which yields a powerful conditional programming construct. Pattern matching sometimes include support for guards.

Term rewriting and graph rewriting languages rely on pattern matching for the fundamental way a program evaluates into a result.

History

The first computer programs to use pattern matching were text editors. At Bell Labs, Ken Thompson extended the seeking and replacing features of the QED editor to accept regular expressions. Early programming languages with pattern matching constructs include SNOBOL from 1962, SASL from 1976, NPL from 1977, and KRC from 1981. The first programming language with tree-based pattern matching features was Fred McBride's extension of LISP, in 1970.^[1]

Primitive patterns

The simplest pattern in pattern matching is an explicit value or a variable. For an example, consider a simple function definition in Haskell syntax (function parameters are not in parentheses but are separated by spaces, = is not assignment but definition):

```
f 0 = 1
```

Here, 0 is a single value pattern. Now, whenever f is given 0 as argument the pattern matches and the function returns 1. With any other argument, the matching fails and thus the function fails. As the syntax supports alternative patterns in function definitions, we can continue the definition extending it to take more generic arguments:

```
f n = n * f (n-1)
```

Here, the first n is a single variable pattern, which will match absolutely any argument and bind it to name n to be used in the rest of the definition. In Haskell (unlike at least Hope), patterns are tried in order so the first definition still applies in the very specific case of the input being 0, while for any other argument the function returns n * f (n-1) with n being the argument.

The wildcard pattern (often written as _) is also simple: like a variable name, it matches any value, but does not bind the value to any name.

Tree patterns

More complex patterns can be built from the primitive ones of the previous section, usually in the same way as values are built by combining other values. The difference then is that with variable and wildcard parts, a pattern doesn't build into a single value, but matches a group of values that are the combination of the concrete elements and the elements that are allowed to vary within the structure of the pattern.

A tree pattern describes a part of a tree by starting with a node and specifying some branches and nodes and leaving some unspecified with a variable or wildcard pattern. It may help to think of the abstract syntax tree of a programming language and algebraic data types.

In Haskell, the following line defines an algebraic data type `Color` that has a single data constructor `ColorConstructor` that wraps an integer and a string.

```
data Color = ColorConstructor Integer String
```

The constructor is a node in a tree and the integer and string are leaves in branches.

When we want to write functions to make `Color` an abstract data type, we wish to write functions to interface with the data type, and thus we want to extract some data from the data type, for example, just the string or just the integer part of `Color`.

If we pass a variable that is of type `Color`, how can we get the data out of this variable? For example, for a function to get the integer part of `Color`, we can use a simple tree pattern and write:

```
integerPart (ColorConstructor theInteger _) = theInteger
```

As well:

```
stringPart (ColorConstructor _ theString) = theString
```

The creation of these functions can be automated by Haskell's data record syntax.

Filtering data with patterns

Pattern matching can be used to filter data of a certain structure. For instance, in Haskell a list comprehension could be used for this kind of filtering:

```
[A x | A x <- [A 1, B 1, A 2, B 2]]
```

evaluates to

```
[A 1, A 2]
```

Pattern matching in Mathematica

In Mathematica, the only structure that exists is the tree, which is populated by symbols. In the Haskell syntax used thus far, this could be defined as

```
data SymbolTree = Symbol String [SymbolTree]
```

An example tree could then look like

```
Symbol "a" [Symbol "b" [], Symbol "c"]
```

In the traditional, more suitable syntax, the symbols are written as they are and the levels of the tree are represented using [], so that for instance `a[b, c]` is a tree with `a` as the parent, and `b` and `c` as the children.

A pattern in Mathematica involves putting `_` at positions in that tree. For instance, the pattern

```
A[_]
```

will match elements such as `A[1]`, `A[2]`, or more generally `A[x]` where `x` is any entity. In this case, `A` is the concrete element, while `_` denotes the piece of tree that can be varied. A symbol prepended to `_` binds the match to that variable name while a symbol appended to `_` restricts the matches to nodes of that symbol.

The Mathematica function `Cases` filters elements of the first argument that match the pattern in the second argument:

```
Cases[{a[1], b[1], a[2], b[2]}, a[_]]
```

evaluates to

```
{a[1], a[2]}
```

Pattern matching applies to the *structure* of expressions. In the example below,

```
Cases[ {a[b], a[b, c], a[b[c], d], a[b[c], de, a[b[c], d, e]}, a[b[_], __] ]
```

returns

```
{a[b[c], d], a[b[c], de]}
```

because only these elements will match the pattern `a[b[_], __]` above.

In Mathematica, it is also possible to extract structures as they are created in the course of computation, regardless of how or where they appear. The function `Trace` can be used to monitor a computation, and return the elements that arise which match a pattern. For example, we can define the Fibonacci sequence as

```
fib[0|1]:=1
fib[n_]:= fib[n-1] + fib[n-2]
```

Then, we can ask the question: Given `fib[3]`, what is the sequence of recursive Fibonacci calls?

```
Trace[fib[3], fib[_]]
```

returns a structure that represents the occurrences of the pattern `fib[_]` in the computational structure:

```
{fib[3],{fib[2],{fib[1]},{{fib[0]}},{{fib[1]}}}}
```

Declarative programming

In symbolic programming languages, it is easy to have patterns as arguments to functions or as elements of data structures. A consequence of this is the ability to use patterns to declaratively make statements about pieces of data and to flexibly instruct functions how to operate.

For instance, the Mathematica function `Compile` can be used to make more efficient versions of the code. In the following example the details do not particularly matter; what matters is that the subexpression `{{{com[_]}, Integer}}}` instructs `Compile` that expressions of the form `com[_]` can be assumed to be integers for the purposes of compilation:

```
com[i_] := Binomial[2i, i]
Compile[{x, {i, _Integer}}, x^com[i], {{com[_], Integer}}]
```

Mailboxes in Erlang also work this way.

The Curry-Howard correspondence between proofs and programs relates ML-style pattern matching to case analysis and proof by exhaustion.

Pattern matching and strings

By far the most common form of pattern matching involves strings of characters. In many programming languages, a particular syntax of strings is used to represent regular expressions, which are patterns describing string characters.

However, it is possible to perform some string pattern matching within the same framework that has been discussed throughout this article.

Tree patterns for strings

In Mathematica, strings are represented as trees of root `StringExpression` and all the characters in order as children of the root. Thus, to match "any amount of trailing characters", a new wildcard `___` is needed in contrast to `_` that would match only a single character.

In Haskell and functional programming languages in general, strings are represented as functional lists of characters. A functional list is defined as an empty list, or an element constructed on an existing list. In Haskell syntax:

```
[] -- an empty list
x:xs -- an element x constructed on a list xs
```

The structure for a list with some elements is thus `element:list`. When pattern matching, we assert that a certain piece of data is equal to a certain pattern. For example, in the function:

```
head (element:list) = element
```

we assert that the first element of `head`'s argument is called `element`, and the function returns this. We know that this is the first element because of the way lists are defined, a single element constructed onto a list. This single element must be the first. The empty list would not match the pattern at all, as an empty list does not have a head (the first element that is constructed).

In the example, we have no use for `list`, so we can disregard it, and thus write the function:

```
head (element:_)= element
```

The equivalent Mathematica transformation is expressed as

```
head[element, ] := element
```

Example string patterns

In Mathematica, for instance,

```
StringExpression["a", ]
```

will match a string that has two characters and begins with "a".

The same pattern in Haskell:

```
['a', _]
```

Symbolic entities can be introduced to represent many different classes of relevant features of a string. For instance,

```
StringExpression[LetterCharacter, DigitCharacter]
```

will match a string that consists of a letter first, and then a number.

In Haskell, guards could be used to achieve the same matches:

```
[letter, digit] | isAlpha letter && isDigit digit
```

The main advantage of symbolic string manipulation is that it can be completely integrated with the rest of the programming language, rather than being a separate, special purpose subunit. The entire power of the language can be leveraged to build up the patterns themselves or analyze and transform the programs that contain them.

SNOBOL

SNOBOL (*String Oriented Symbolic Language*) is a computer programming language developed between 1962 and 1967 at AT&T Bell Laboratories by David J. Farber, Ralph E. Griswold and Ivan P. Polonsky.

SNOBOL4 stands apart from most programming languages by having patterns as a first-class data type (*i.e.* a data type whose values can be manipulated in all ways permitted to any other data type in the programming language) and by providing operators for pattern concatenation and alternation. Strings generated during execution can be treated as programs and executed.

SNOBOL was quite widely taught in larger US universities in the late 1960s and early 1970s and was widely used in the 1970s and 1980s as a text manipulation language in the humanities.

Since SNOBOL's creation, newer languages such as Awk and Perl have made string manipulation by means of regular expressions fashionable. SNOBOL4 patterns, however, subsume BNF grammars, which are equivalent to context-free grammars and more powerful than regular expressions ^[2]

References

- The Mathematica Book, chapter Section 2.3: Patterns [3]
- The Haskell 98 Report, chapter 3.17 Pattern Matching [4].
- Python Reference Manual, chapter 6.3 Assignment statements [5].
- The Pure Programming Language, chapter 4.3: Patterns [6]

[1] <http://www.cs.nott.ac.uk/~ctm/view.ps.gz>

[2] Gimpel, J. F. 1973. A theory of discrete patterns and their implementation in SNOBOL4. Commun. ACM 16, 2 (Feb. 1973), 91-100.
DOI=<http://doi.acm.org/10.1145/361952.361960>

[3] <http://documents.wolfram.com/mathematica/book/section-2.3>

[4] <http://haskell.org/onlinereport/expo.html#pattern-matching>

[5] <http://python.org/doc/2.4.1/ref/assignment.html>

[6] <http://pure-lang.googlecode.com/svn/docs/pure-intro/pure-intro.pdf>

External links

- A Gentle Introduction to Haskell: Patterns (<http://www.haskell.org/tutorial/patterns.html>)
- Views: An Extension to Haskell Pattern Matching (<http://www.haskell.org/development/views.html>)
- Nikolaas N. Oosterhof, Philip K. F. Hölzenspies, and Jan Kuper. Application patterns (<http://wwwhome.cs.utwente.nl/~tina/apm/applPatts.pdf>). A presentation at Trends in Functional Programming, 2005
- JMatch (<http://www.cs.cornell.edu/Projects/jmatch>): the Java programming language extended with pattern matching
- An incomplete history of the QED Text Editor (<http://cm.bell-labs.com/cm/cs/who/dmr/qed.html>) by Dennis Ritchie - provides the history of regular expressions in computer programs
- The Implementation of Functional Programming Languages, pages 53-103 (<http://research.microsoft.com/~simonpj/papers/slpj-book-1987/index.htm>) Simon Peyton Jones, published by Prentice Hall, 1987.
- Nemerle, pattern matching (http://nemerle.org/Grok_Variants_and_matching#Matching).
- Erlang, pattern matching (http://erlang.org/doc/reference_manual/expressions.html#pattern).
- Prop: a C++ based pattern matching language, 1999 (<http://www.cs.nyu.edu/leunga/prop.html>)
- Temur Kutsia. Flat Matching (<http://dx.doi.org/10.1016/j.jsc.2008.05.001>). Journal of Symbolic Computation 43(12):858—873. Describes in details flat matching in Mathematica.

Anonymous function

In computer programming, an **anonymous function** (also **function constant**, **function literal**, or **lambda function**) is a function (or a subroutine) defined, and possibly called, without being bound to an identifier. Anonymous functions are convenient to pass as an argument to a higher-order function and are ubiquitous in languages with first-class functions such as Haskell. Anonymous functions are a form of nested function, in that they allow access to the variable in the scope of the containing function (non-local variables). Unlike named nested functions, they cannot be recursive without the assistance of a fixpoint operator (also known as an *anonymous fixpoint* or *anonymous recursion*).

Anonymous functions originate in the work of Alonzo Church in his invention of the lambda calculus in 1936 (prior to electronic computers), in which all functions are anonymous. In several programming languages, anonymous functions are introduced using the keyword **lambda**, and anonymous functions are often referred to as lambda functions.

Anonymous functions have been a feature of programming languages since Lisp in 1958. An increasing number of modern programming languages support anonymous functions, and some notable mainstream languages have recently added support for them, the most widespread being JavaScript,^[1] C#,^[2] Ruby^[3] and PHP^[4]. Anonymous functions were added to C++ in C++11. Some object-oriented programming languages have anonymous classes, which are a similar concept, but do not support anonymous functions. Java is such a language (although support for lambdas is on the roadmap for Java 8^[5]).

Uses

Anonymous functions can be used to contain functionality that need not be named and possibly for short-term use. Some notable examples include closures and currying.

All of the code in the following sections is written in Python 2.x (not 3.x).

Sorting

When attempting to sort in a non-standard way it may be easier to contain the comparison logic as an anonymous function instead of creating a named function. Most languages provide a generic sort function that implements a sort algorithm that will sort arbitrary objects. This function usually accepts an arbitrary comparison function that is supplied two items and the function indicates if they are equal or if one is "greater" or "less" than the other (typically indicated by returning a negative number, zero, or a positive number).

Consider sorting items in a list by the name of their class (in Python, everything has a class):

```
a = [10, '10', 10.0]
a.sort(lambda x,y: cmp(x.__class__.__name__, y.__class__.__name__))
print a
[10.0, 10, '10']
```

Note that 10.0 has class name "float", 10 has class name "int", and '10' has class name "str". The sorted order is "float", "int", then "str".

The anonymous function in this example is the lambda expression:

```
lambda x,y: cmp(...)
```

The anonymous function accepts two arguments, x and y, and returns the comparison between them using the built-in function `cmp()`. Another example would be sorting a list of strings by length of the string:

```
a = ['three', 'two', 'four']
a.sort(lambda x,y: cmp(len(x), len(y)))
print a
['two', 'four', 'three']
```

which clearly has been sorted by length of the strings.

Closures

Closures are functions evaluated in an environment containing bound variables. The following example binds the variable "threshold" in an anonymous function that compares the input to the threshold.

```
def comp(threshold):
    return lambda x: x < threshold
```

This can be used as a sort of generator of comparison functions:

```
a = comp(10)
b = comp(20)

print a(5), a(8), a(13), a(21)
True True False False

print b(5), b(8), b(13), b(21)
True True True False
```

It would be very impractical to create a function for every possible comparison function and may be too inconvenient to keep the threshold around for further use. Regardless of the reason why a closure is used, the anonymous function is the entity that contains the functionality that does the comparing.

Currying

Currying is transforming a function from multiple inputs to fewer inputs (in this case integer division).

```
def divide(x,y):
    return x/y

def divisor(d):
    return lambda x: divide(x,d)

half = divisor(2)
third = divisor(3)

print half(32), third(32)
16 10

print half(40), third(40)
20 13
```

While the use of anonymous functions is perhaps not common with currying it still can be used. In the above example, the function divisor generates functions with a specified divisor. The functions half and third curry the divide function with a fixed divisor.

(It just so happens that the divisor function forms a closure as well as curries by binding the "d" variable.)

Higher-order functions

Map

The map function performs a function call on each element of a list. The following example squares every element in an array with an anonymous function.

```
a = [1, 2, 3, 4, 5, 6]
print map(lambda x: x*x, a)
[1, 4, 9, 16, 25, 36]
```

The anonymous function accepts an argument and multiplies it by itself (squares it).

Filter

The filter function returns all elements from a list that evaluate True when passed to a certain function.

```
a = [1, 2, 3, 4, 5, 6]
print filter(lambda x: x % 2 == 0, a)
[2, 4, 6]
```

The anonymous function checks if the argument passed to it is even.

Fold

The fold/reduce function runs over all elements in a list (usually left-to-right), accumulating a value as it goes. A common usage of this is to combine all elements of a list into a single value, for example:

```
a = [1, 2, 3, 4, 5]
print reduce(lambda x,y: x*y, a)
120
```

This performs:

$$(((1 \times 2) \times 3) \times 4) \times 5 = 120$$

The anonymous function here is simply the multiplication of the two arguments.

However, there is no reason why the result of a fold need be a single value - in fact, both map and filter can be created using fold. In map, the value that is accumulated is a new list, containing the results of applying a function to each element of the original list. In filter, the value that is accumulated is a new list containing only those elements that match the given condition.

List of languages

The following is a list of programming languages that fully support unnamed anonymous functions; support some variant of anonymous functions; and have no support for anonymous functions.

This table shows some general trends. First, the languages that do not support anonymous functions—C, Pascal, Object Pascal, Java—are all conventional statically typed languages. This does not, however, mean that statically typed languages are incapable of supporting anonymous functions. For example, the ML languages are statically typed and fundamentally include anonymous functions, and Delphi, a dialect of Object Pascal, has been extended to support anonymous functions. Second, the languages that treat functions as first-class functions—Dylan, JavaScript, Lisp, Scheme, ML, Haskell, Python, Ruby, Perl—generally have anonymous function support so that functions can be defined and passed around as easily as other data types. However, the new C++11 standard adds them to C++,

even though this is a conventional, statically typed language.

This list is incomplete.

Language	Support	Notes
ActionScript	✓	
C	✗	Support is provided in clang and along with the llvm compiler-rt lib. GCC support is given for a macro implementation which enables the possibility of usage. Details see below.
C#	✓	
C++	✓	As of the C++11 standard
Clojure	✓	
Curl	✓	
D	✓	
Dart	✓	
Dylan	✓	
Erlang	✓	
F#	✓	
Frink	✓	
Go	✓	
Groovy	✓ ^[6]	
Haskell	✓	
Java	✗	Planned for Java 8
JavaScript	✓	
Lisp	✓	
Logtalk	✓	
Lua	✓	
Mathematica	✓	
Maple	✓	
Matlab	✓	
Maxima	✓	
ML languages (OCaml, Standard ML, etc.)	✓	
Octave	✓	
Object Pascal	✓	Delphi, a dialect of Object Pascal, implements support for anonymous functions (formally, <i>anonymous methods</i>) natively since Delphi 2009. The Oxygene Object Pascal dialect also supports them.
Objective-C (Mac OS X 10.6+)	✓	called blocks; in addition to Objective-C, blocks can also be used on C and C++ when programming on Apple's platform
Pascal	✗	
Perl	✓	
PHP	✓	As of PHP 5.3.0, true anonymous functions are supported; previously only partial anonymous functions were supported, which worked much like C#'s implementation.

Python	✓	Python supports anonymous functions through the lambda syntax, in which you can only use expressions (and not statements).
R	✓	
Racket	✓	
Ruby	✓	Ruby's anonymous functions, inherited from Smalltalk, are called blocks.
Scala	✓	
Scheme	✓	
Smalltalk	✓	Smalltalk's anonymous functions are called blocks.
TypeScript	✓	
Tcl	✓	
Vala	✓	
Visual Basic .NET v9	✓	
Visual Prolog v 7.2	✓	

Examples

Numerous languages support anonymous functions, or something similar.

C lambda expressions

The following example works only with GCC. Also note that due to the way macros work, if your l_body contains any commas outside of parentheses then it will not compile as gcc uses the comma as a delimiter for the next argument in the macro. The argument 'l_ret_type' can be removed if '__typeof__' is available to you; in the example below using __typeof__ on array would return testtype *, which can be dereferenced for the actual value if needed.

```
/* this is the definition of the anonymous function */
#define lambda(l_ret_type, l_arguments, l_body) \
({ \
    l_ret_type l_anonymous_functions_name l_arguments \
    l_body \
    &l_anonymous_functions_name; \
})

#define forEachInArray(fe_arrType, fe_arr, fe_fn_body) \
{ \
    \
    int i=0; \
    \
    for(;i<sizeof(fe_arr)/sizeof(fe_arrType);i++) { fe_arr[i] = fe_fn_body(&fe_arr[i]); } \
}

typedef struct __test {
    int a;
```

```

    int b;
} testtype;

testtype array[] = { {0,1}, {2,3}, {4,5} };

printout(array);

/* the anonymous function is given as function for the foreach */
forEachInArray(testtype, array,
    lambda (testtype, (void *item),
    {
        int temp = (* ( testtype *) item).a;
        (* ( testtype *) item).a = (* ( testtype *) item).b;
        (* ( testtype *) item).b = temp;
        return (* ( testtype *) item);
    }));
printout(array);

```

Using the aforementioned clang extension called **block** and libdispatch, Code could look simpler like this:

```

#include <stdio.h>
#include <dispatch.h>

int main(void)
{
    void (^f) () = ^{
        for (int i = 0; i < 100; i++)
            printf("%d\n", i);
    } //make a variable of it

    dispatch_async(dispatch_get_global_queue(DISPATCH_QUEUE_PRIORITY_DEFAULT,
        0), f); //Pass as a parameter to another function, in this case
    libdispatch
        f(); //Direct invoke
        return 0;
}

```

C# lambda expressions

Support for anonymous functions in C# has deepened through the various versions of the language compiler. The C# language v3.0, released in November 2007 with the .NET Framework v3.5, has full support of anonymous functions. C# refers to them as "lambda expressions", following the original version of anonymous functions, the lambda calculus. See the C# 4.0 Language Specification^[7], section 5.3.3.29, for more information.

```

// the first int is the x' type
// the second int is the return type
// <see href="http://msdn.microsoft.com/en-us/library/bb549151.aspx" />
Func<int, int> foo = x => x*x;
Console.WriteLine(foo(7));

```

While the function is anonymous, it cannot be assigned to an implicitly typed variable, because the lambda syntax may be used to denote an anonymous function or an expression tree, and the choice cannot automatically be decided by the compiler. E.g., this does not work:

```
// will NOT compile!
var foo = (int x) => x*x;
```

However, a lambda expression can take part in type inference and can be used as a method argument, e.g. to use anonymous functions with the Map capability available with `System.Collections.Generic.List` (in the `ConvertAll()` method):

```
// Initialize the list:
var values = new List<int>() { 7, 13, 4, 9, 3 };
// Map the anonymous function over all elements in the list, return the
new list
var foo = values.ConvertAll(d => d*d) ;
// the result of the foo variable is of type
System.Collections.Generic.List<Int32>
```

Prior versions of C# had more limited support for anonymous functions. C# v1.0, introduced in February 2002 with the .NET Framework v1.0, provided partial anonymous function support through the use of delegates. This construct is somewhat similar to PHP delegates. In C# 1.0, Delegates are like function pointers that refer to an explicitly named method within a class. (But unlike PHP the name is not required at the time the delegate is used.) C# v2.0, released in November 2005 with the .NET Framework v2.0, introduced the concept of anonymous methods as a way to write unnamed inline statement blocks that can be executed in a delegate invocation. C# 3.0 continues to support these constructs, but also supports the lambda expression construct.

This example will compile in C# 3.0, and exhibits the three forms:

```
public class TestDriver
{
    delegate int SquareDelegate(int d);
    static int Square(int d)
    {
        return d*d;
    }

    static void Main(string[] args)
    {
        // C# 1.0: Original delegate syntax required
        // initialization with a named method.
        SquareDelegate A = new SquareDelegate(Square);
        System.Console.WriteLine(A(3));

        // C# 2.0: A delegate can be initialized with
        // inline code, called an "anonymous method." This
        // method takes an int as an input parameter.
        SquareDelegate B = delegate(int d) { return d*d; };
        System.Console.WriteLine(B(5));

        // C# 3.0. A delegate can be initialized with
    }
}
```

```

    // a lambda expression. The lambda takes an int, and returns an
    int.

    // The type of x is inferred by the compiler.
    SquareDelegate C = x => x*x;
    System.Console.WriteLine(C(7));

    // C# 3.0. A delegate that accepts a single input and
    // returns a single output can also be implicitly declared with
    the Func<> type.
    System.Func<int,int> D = x => x*x;
    System.Console.WriteLine(D(9));
}
}

```

In the case of the C# 2.0 version, the C# compiler takes the code block of the anonymous function and creates a static private function. Internally, the function gets a generated name, of course; this generated name is based on the name of the method in which the Delegate is declared. But the name is not exposed to application code except by using reflection.

In the case of the C# 3.0 version, the same mechanism applies.

C++

C++11 provides support for anonymous functions, called lambda functions in the specification. A lambda expression has the form:

```
[capture] (parameters) ->return-type {body}
```

If there are no parameters the empty parentheses can be omitted. The return type can often be omitted, if the body consists only of one return statement or the return type is void.

```
[capture] (parameters) {body}
```

An example lambda function is defined as follows:

```
[](int x, int y) { return x + y; } // implicit return type from
'<b>return</b>' statement
[](int& x) { ++x; } // no return statement -> lambda functions' return
type is 'void'
[]() { ++global_x; } // no parameters, just accessing a global
variable
[]{ ++global_x; } // the same, so () can be omitted
```

The return type of this unnamed function is `decltype(x+y)`. The return type can be omitted if the lambda function is of the form `return expression` (or if the lambda returns nothing), or if all locations that return a value return the same type when the return expression is passed through `decltype`.

The return type can be explicitly specified as follows:

```
[](int x, int y) -> int { int z = x + y; return z; }
```

In this example, a temporary variable, `z`, is created to store an intermediate. As with normal functions, the value of this intermediate is not held between invocations. Lambdas that return nothing can omit the return type specification;

they do not need to use `-> void`.

A lambda function can refer to identifiers declared outside the lambda function. The set of these variables is commonly called a closure. Closures are defined between square brackets [and] in the declaration of lambda expression. The mechanism allows these variables to be captured by value or by reference. The following table demonstrates this:

<code>[]</code>	<i>//no variables defined. Attempting to use any external variables in the lambda is an error.</i>
<code>[x, &y]</code>	<i>//x is captured by value, y is captured by reference</i>
<code>[&]</code>	<i>//any external variable is implicitly captured by reference</i>
<code>if</code> used	
<code>[=]</code>	<i>//any external variable is implicitly captured by value if used</i>
<code>[&, x]</code>	<i>//x is explicitly captured by value. Other variables will be captured by reference</i>
<code>[=, &z]</code>	<i>//z is explicitly captured by reference. Other variables will be captured by value</i>

The following two examples demonstrate usage of a lambda expression:

```
std::vector<int> some_list;
int total = 0;
for (int i=0;i<5;++i) some_list.push_back(i);
std::for_each(begin(some_list), end(some_list), [&total](int x) {
    total += x;
});
```

This computes the total of all elements in the list. The variable `total` is stored as a part of the lambda function's closure. Since it is a reference to the stack variable `total`, it can change its value.

```
std::vector<int> some_list;
int total = 0;
int value = 5;
std::for_each(begin(some_list), end(some_list), [&, value, this](int x)
{
    total += x * value * this->some_func();
});
```

This will cause `total` to be stored as a reference, but `value` will be stored as a copy.

The capture of `this` is special. It can only be captured by value, not by reference. `this` can only be captured if the closest enclosing function is a non-static member function. The lambda will have the same access as the member that created it, in terms of protected/private members.

If `this` is captured, either explicitly or implicitly, then the scope of the enclosed class members is also tested. Accessing members of `this` does not require explicit use of `this->` syntax.

The specific internal implementation can vary, but the expectation is that a lambda function that captures everything by reference will store the actual stack pointer of the function it is created in, rather than individual references to stack variables. However, because most lambda functions are small and local in scope, they are likely candidates for inlining, and thus will not need any additional storage for references.

If a closure object containing references to local variables is invoked after the innermost block scope of its creation, the behaviour is undefined.

Lambda functions are function objects of an implementation-dependent type; this type's name is only available to the compiler. If the user wishes to take a lambda function as a parameter, the type must be a template type, or they must create a `std::function` or a similar object to capture the lambda value. The use of the `auto` keyword can help store the lambda function,

```
auto my_lambda_func = [&] (int x) { /*...*/ };
auto my_onheap_lambda_func = new auto([=] (int x) { /*...*/ });
```

Here is an example of storing anonymous functions in variables, vectors, and arrays; and passing them as named parameters:

```
#include<functional>
#include<vector>
#include<iostream>
double eval(std::function<double(double)> f, double x = 2.0) {return f(x);}
int main() {
    std::function<double(double)> f0      = [] (double x){return 1;};
    auto                      f1      = [] (double x){return x;};
    decltype(f0)                 fa[3] = {f0,f1,[ ] (double x){return
x*x; }};
    std::vector<decltype(f0)>     fv      = {f0,f1};
    fv.push_back                ([] (double x){return x*x; });
    for(int i=0;i<fv.size();i++) std::cout << fv[i](2.0) << "\n";
    for(int i=0;i<3;i++)        std::cout << fa[i](2.0) << "\n";
    for(auto &f : fv)           std::cout << f(2.0) << "\n";
    for(auto &f : fa)           std::cout << f(2.0) << "\n";
    std::cout << eval(f0) << "\n";
    std::cout << eval(f1) << "\n";
    return 0;
}
```

A lambda function with an empty capture specification (`[]`) can be implicitly converted into a function pointer with the same type as the lambda was declared with. So this is legal:

```
auto a_lambda_func = [] (int x) { /*...*/ };
void(*func_ptr)(int) = a_lambda_func;
func_ptr(4); //calls the lambda.
```

D

```
(x) {return x*x;}
delegate (x) {return x*x;} // if more verbosity is needed
(int x) {return x*x;} // if parameter type cannot be inferred
delegate (int x){return x*x;} // ditto
delegate double(int x){return x*x;} // if return type must be forced
manually
```

Since version 2.0, D allocates closures on the heap unless the compiler can prove it is unnecessary; the `scope` keyword can be used to force stack allocation. Since version 2.058, it is possible to use shorthand notation:

```
x => x*x;
(int x) => x*x;
(x,y) => x*y;
(int x, int y) => x*y;
```

Dart

Dart supports anonymous functions.

```
var sqr = (x) => x * x;
print(sqr(5));
```

or

```
print(((x) => x * x)(5));
```

Delphi (since v. 2009)

```
program demo;

type
  TSimpleProcedure = reference to procedure;
  TSimpleFunction = reference to function(x: string): Integer;

var
  x1: TSimpleProcedure;
  y1: TSimpleFunction;

begin
  x1 := procedure
    begin
      Writeln('Hello World');
    end;
  x1; //invoke anonymous method just defined

  y1 := function(x: string): Integer
    begin
      Result := Length(x);
    end;
  Writeln(y1('bar'));
end.
```

Erlang

Erlang uses a syntax for anonymous functions similar to that of named functions.

```
% Anonymous function bound to the Square variable
Square = fun(X) -> X * X end.

% Named function with the same functionality
square(X) -> X * X.
```

Haskell

Haskell uses a concise syntax for anonymous functions (lambda expressions).

```
\x -> x * x
```

Lambda expressions are fully integrated with the type inference engine, and support all the syntax and features of "ordinary" functions (except for the use of multiple definitions for pattern-matching, since the argument list is only specified once).

```
map (\x -> x * x) [1..5] -- returns [1, 4, 9, 16, 25]
```

The following are all equivalent:

```
f x y = x + y
f x = \y -> x + y
f = \x y -> x + y
```

JavaScript

JavaScript supports anonymous functions.

```
alert((function(x) {
    return x*x;
}) (10));
```

This construct is often used in Bookmarklets. For example, to change the title of the current document (visible in its window's title bar) to its URL, the following bookmarklet may seem to work.

```
javascript:document.title=location.href;
```

However, as the assignment statement returns a value (the URL itself), many browsers actually create a new page to display this value.

Instead, an anonymous function, that does not return a value, can be used:

```
javascript:(function() {document.title=location.href;})();
```

The function statement in the first (outer) pair of parentheses declares an anonymous function, which is then executed when used with the last pair of parentheses. This is almost equivalent to the following, which populates the environment with `f` unlike an anonymous function.

```
javascript:var f = function() {document.title=location.href;}; f();
```

Use `void()` to avoid new pages for arbitrary anonymous functions:

```
javascript:void(function() {return document.title=location.href;})();
```

(or just:

```
javascript:void(document.title=location.href);
)
```

Javascript has syntactic subtleties for the semantics of defining, invoking and evaluating anonymous functions. These subliminal nuances are a direct consequence of the evaluation of parenthetical expressions. The following constructs illustrate this:

```
(function() { ... } ())
```

and

```
(function() { ... })()
```

. Representing "function () { ... }" by f , the form of the constructs are a parenthetical within a parenthetical ($f()$) and a parenthetical applied to a parenthetical ($f()$).

Note the general syntactic ambiguity of a parenthetical expression, parenthesized arguments to a function and the parentheses around the formal parameters in a function definition. In particular, Javascript defines a , (comma) operator in the context of a parenthetical expression. It is no mere coincidence that the syntactic forms coincide for an expression and a function's arguments (ignoring the function formal parameter syntax)! If f is not identified in the constructs above, they become $(())$ and $()()$. The first provides no syntactic hint of any resident function but the second MUST evaluate the first parenthetical as a function to be legal Javascript. (Aside: for instance, the $()$'s could be $([], {}, 42, "abc", function() {})$ as long as the expression evaluates to a function.)

Also, a function is an Object instance (likewise objects are Function instances) and the object literal notation brackets, $\{ \}$ for braced code, are used when defining a function this way (as opposed to using `new Function(...)`). In a very broad non-rigorous sense (especially since global bindings are compromised), an arbitrary sequence of braced Javascripts statements, $\{stuff\}$, can be considered to be a fixed point of

```
(function() { ( function() { ( ... { ( function() {stuff} () ) } ... ) } () ) } () )
```

More correctly but with caveats,

```
( function() {stuff} () ) ~=
A_Fixed_Point_of(
    function(){ return function(){ return ... { return
function() {stuff} () ... } () } () }
)
```

Note the implications of the anonymous function in the javascripts that follow:

```
(["Using: ",window.navigator.userAgent] );
/*
Using: ,Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:11.0) Gecko/20100101 Firefox/11.0
with OUT the Unity desktop!!
*/
```

- `function() { ... }()` without surrounding $()$'s is generally not legal
- `(f=function() { ... })` does not "forget" f globally unlike `(function f() { ... })`
- "relaxed" function "naming" such as:

```
RA[3]=function() { ... }
```

or even this bizarre "name" creation

```
eval(          "f:j=function(x){           alert((x))           }"        );
eval("f:j") (eval("f:j"));
(this simply prints function(x){ alert((x)) } )
```

- objects can have function property identifiers (ie. be "indexed" by a function):

```
({} [function() {}])=42
alert(function      (obj){      obj[function() {}]=42;      return
obj}({}) [function() {}]) /* 42 */
```

- `(function(f,x){return (f)(x)})` is a particularly "potent" lambda potion

- (function (l){return (l)(l)})(function (l){return (l)(l)}) ie. (lambda)(lambda)

 (function (l){return (l(l))} (function (l){return (l(l))}))
- (function (i){return (i)(i)(i)(i) ... (i)(i)})(function (f){return f})
- (function (r){return (r(r(r(r(...(r)...)))))})(function (f){return f})
- (function (r){return (((...(((r)(r))(r))...(r))(r))(r))})(function (f){return f})

Performance metrics to analyze the space and time complexities of function calls, call stack, etc. in a Javascript interpreter engine implement easily with these last anonymous function constructs. From the implications of the results, it is possible to deduce some of an engine's recursive versus iterative implementation details, especially tail-recursion .

Javascript is a functional programming language and a lambda calculus in a profoundly pure sense, providing particularly pristine programming for recursive function theory paradigms.

Lisp

Lisp and Scheme support anonymous functions using the "lambda" construct, which is a reference to lambda calculus. Clojure supports anonymous functions with the "fn" special form and #() reader syntax.

```
(lambda (arg) (* arg arg))
```

Interestingly, Scheme's "named functions" is simply syntactic sugar for anonymous functions bound to names:

```
(define (somename arg)
  (do-something arg))
```

expands (and is equivalent) to

```
(define somename
  (lambda (arg)
    (do-something arg)))
```

Clojure supports anonymous functions through the "fn" special form:

```
(fn [x] (+ x 3))
```

There is also a reader syntax to define a lambda:

```
# (+ % %2 %3) ; Defines an anonymous function that takes three
arguments and sums them.
```

Like Scheme, Clojure's "named functions" are simply syntactic sugar for lambdas bound to names:

```
(defn func [arg] (+ 3 arg))
```

expands to:

```
(def func (fn [arg] (+ 3 arg)))
```

Logtalk

Logtalk uses the following syntax for anonymous predicates (lambda expressions):

```
{FreeVar1, FreeVar2, ...} / [LambdaParameter1, LambdaParameter2, ...] >> Goal
```

A simple example with no free variables and using a list mapping predicate is:

```
| ?- meta::map([X,Y]>>(Y is 2*X), [1,2,3], Ys).  
Ys = [2,4,6]  
yes
```

Currying is also supported. The above example can be written as:

```
| ?- meta::map([X]>>([Y]>>(Y is 2*X)), [1,2,3], Ys).  
Ys = [2,4,6]  
yes
```

Lua

In Lua (much as in Scheme) all functions are anonymous. A "named function" in Lua is simply a variable holding a reference to a function object.^[8]

Thus, in Lua

```
function foo(x) return 2*x end
```

is just syntactical sugar for

```
foo = function(x) return 2*x end
```

An example of using anonymous functions for reverse-order sorting:

```
table.sort(network, function(a,b)  
    return a.name > b.name  
end)
```

Mathematica

Anonymous Functions are important in programming Mathematica. There are several ways to create them. Below are a few anonymous function that increment a number. The first is the most common. '#1' refers to the first argument and '&' makes the end of the anonymous function.

```
#1+1&  
Function[x,x+1]  
x \[Function] x+1
```

Additionally, Mathematica has an additional construct to for making recursive anonymous functions. The symbol '#0' refers to the entire function.

```
If[#1 == 1, 1, #1 * #0[#1-1]]&
```

Maxima

In Maxima anonymous functions are defined using the syntax `lambda(argument-list,expression)`,

```
f: lambda([x],x*x); f(8);
64

lambda([x,y],x+y) (5,6);
11
```

ML

The various dialects of ML support anonymous functions.

OCaml:

```
fun arg -> arg * arg
```

F#:

```
(fun x -> x * x) 20 // 400
```

Standard ML:

```
fn arg => arg * arg
```

Octave

Anonymous functions in GNU Octave are defined using the syntax `@(argument-list)expression`. Any variables that are not found in the argument list are inherited from the enclosing scope.

```
octave:1> f = @(x)x*x; f(8)
ans = 64
octave:2> @(x,y)x+y) (5,6)
ans = 11
```

Perl

Perl 5

Perl 5 supports anonymous functions, as follows:

```
(sub { print "I got called\n" })->(); # 1. fully anonymous,
called as created

my $squarer = sub { my $x = shift; $x * $x }; # 2. assigned to a
variable

sub curry {
  my ($sub, @args) = @_;
  return sub { $sub->(@args, @_)}; # 3. as a return value of
another function
}

# example of currying in Perl
sub sum { my $tot = 0; $tot += $_ for @_; $tot } # returns the sum of
```

```
its arguments
my $curried = curry \&sum, 5, 7, 9;
print $curried->(1,2,3), "\n";      # prints 27 (= 5 + 7 + 9 + 1 + 2 + 3
)
```

Other constructs take "bare blocks" as arguments, which serve a function similar to lambda functions of a single parameter, but don't have the same parameter-passing convention as functions -- @_ is not set.

```
my @squares = map { $_[ * $_[ } 1..10;    # map and grep don't use the
'sub' keyword
my @square2 = map $_[ * $_[, 1..10;        # parentheses not required for a
single expression

my @bad_example = map { print for @_ } 1..10; # values not passed like
normal Perl function
```

Perl 6

In Perl 6, all blocks (even the ones associated with if, while, etc.) are anonymous functions. A block that is not used as an rvalue is executed immediately.

```
{ say "I got called" };                # 1. fully anonymous, called as
created

my $squarer1 = -> $x { $x * $x };          # 2a. assigned to a
variable, pointy block
my $squarer2 = { $^x * $^x };              # 2b. assigned to a
variable, twigil
my $squarer3 = { my $x = shift @_; $x * $x }; # 2b. assigned to a
variable, Perl 5 style

# 3 currying
sub add ($m, $n) { $m + $n }
my $seven = add(3, 4);
my $add_one = &add.assuming(m => 1);
my $eight = $add_one($seven);
```

PHP

Prior to 4.0.1, PHP had no anonymous function support.^[9]

4.0.1 to 5.3

PHP 4.0.1 introduced the `create_function` which was the initial anonymous function support. This function call creates a new randomly named function and returns its name (as a string)

```
$foo = create_function('$x', 'return $x*$x;');
$bar = create_function("\$x", "return \$x*\$x;");
echo $foo(10);
```

It is important to note that the argument list and function body must be in single quotes or the dollar signs must be escaped. Otherwise PHP will assume "\$x" means the variable \$x and will substitute it into the string (despite

possibly not existing) instead of leaving "\$x" in the string. For functions with quotes or functions with lots of variables, it can get quite tedious to ensure the intended function body is what PHP interprets.

It must also be noted that each invocation of `create_function` will create a new function which exists for the rest of the program, and cannot be "garbage collected". If one uses this to create anonymous functions many times, e.g. in a loop, it will irreversibly use up memory in the program.

5.3

PHP 5.3 added a new class called `Closure` and magic method `__invoke()` that makes a class instance invocable.^[10] Lambda functions are a compiler "trick"^[11] that instantiates a new `Closure` instance that can be invoked as if the function were invokable.

```
$x = 3;
$func = function($z) { return $z *= 2; };
echo $func($x); // prints 6
```

In this example, `$func` is an instance of `Closure` and `echo $func()` is equivalent to `$func->__invoke($z)`. PHP 5.3 mimics anonymous functions but it does not support true anonymous functions because PHP functions are still not first-class objects.

PHP 5.3 does support closures but the variables must be explicitly indicated as such:

```
$x = 3;
$func = function() use(&$x) { $x *= 2; };
$func();
echo $x; // prints 6
```

The variable `$x` is bound by reference so the invocation of `$func` modifies it and the changes are visible outside of the function.

Python

Python supports simple anonymous functions through the `lambda` form. The executable body of the lambda must be an expression and can't be a statement, which is a restriction that limits its utility. The value returned by the lambda is the value of the contained expression. Lambda forms can be used anywhere ordinary functions can, however these restrictions make it a very limited version of a normal function. Here is an example:

```
foo = lambda x: x*x
print foo(10)
```

This example will print: 100.

In general, Python convention encourages the use of named functions defined in the same scope as one might typically use an anonymous functions in other languages. This is acceptable as locally defined functions implement the full power of closures and are almost as efficient as the use of a lambda in Python. In this example, the built-in power function can be said to have been curried:

```
def make_pow(n):
    def fixed_exponent_pow(x):
        return pow(x, n)
    return fixed_exponent_pow
sqr = make_pow(2)
print sqr(10) # Emits 100
```

```
cub = make_pow(3)
print cub(10) # Emits 1000
```

R

In GNU R the anonymous functions are defined using the syntax `function(argument-list) expression` .

```
f <- function(x)x*x; f(8)
64
(function(x,y)x+y)(5,6)
11
```

Ruby

Ruby supports anonymous functions by using a syntactical structure called *block*. When passed to a method, a block is converted into an object of class *Proc* in some circumstances.

```
# Example 1:
# Purely anonymous functions using blocks.
ex = [16.2, 24.1, 48.3, 32.4, 8.5]
ex.sort_by { |x| x - x.to_i } # sort by fractional part, ignoring
integer part.
# [24.1, 16.2, 48.3, 32.4, 8.5]

# Example 2:
# First-class functions as an explicit object of Proc -
ex = Proc.new { puts "Hello, world!" }
ex.call # Hello, world!

# Example 3:
# Function that returns lambda function object with parameters
def is_multiple_of(n)
    lambda{|x| x % n == 0}
end
multiple_four = is_multiple_of(4)
multiple_four.call(16)
#true
multiple_four[15]
#false
```

Scala

In Scala, anonymous functions use the following syntax^[12]:

```
(x: Int, y: Int) => x + y
```

In certain contexts, such as when an anonymous function is passed as a parameter to another function, the compiler can infer the types of the parameters of the anonymous function and they can be omitted in the syntax. In such contexts, it is also possible to use a shorthand for anonymous functions using the underscore character to introduce unnamed parameters.

```

val list = List(1, 2, 3, 4)
list.reduceLeft( (x, y) => x + y )
// Here, the compiler can infer that the types of x and y are both Int.

// Therefore, it does not require type annotations on the parameters of
the anonymous function.

list.reduceLeft( _ + _ )
// Each underscore stands for a new unnamed parameter in the anonymous
function.
// This results in an even shorter equivalent to the anonymous function
above.

```

Smalltalk

In Smalltalk anonymous functions are called blocks

```
[ :x | x*x ] value: 2
"returns 4"
```

Tcl

In Tcl, applying the anonymous squaring function to 2 looks as follows:^[13]

```
apply {x {expr {$x*$x}}}
# returns 4
```

It should be observed that this example involves two candidates for what it means to be a "function" in Tcl. The most generic is usually called a *command prefix*, and if the variable *f* holds such a function, then the way to perform the function application *f(x)* would be

```
{*} $f $x
```

where {*} is the expansion prefix (new in Tcl 8.5). The command prefix in the above example is **apply** {x {**expr** {\$x*\$x}}}. Command names can be bound to command prefixes by means of the **interp alias** command. Command prefixes support currying. Command prefixes are very common in Tcl APIs.

The other candidate for "function" in Tcl is usually called a *lambda*, and appears as the {x {**expr** {\$x*\$x}}} part of the above example. This is the part which caches the compiled form of the anonymous function, but it can only be invoked by being passed to the **apply** command. Lambdas do not support currying, unless paired with an **apply** to form a command prefix. Lambdas are rare in Tcl APIs.

Visual Basic

Visual Basic 2008, introduced in November 2007, supports anonymous functions through the lambda form. Combined with implicit typing, VB provides an economical syntax for anonymous functions. As with Python, in VB, anonymous functions must be defined on a single line; they cannot be compound statements. Further, an anonymous function in VB must truly be a VB "Function" - it must return a value.

```
Dim foo = Function(x) x * x
Console.WriteLine(foo(10))
```

Visual Basic 2010, released April 12, 2010, added support for multiline lambda expressions and anonymous functions without a return value. For example, a function for use in a Thread.

```

Dim t As New System.Threading.Thread(Sub()
    For n as Integer = 0 to 10      'Count to 10
        Console.WriteLine(n)         'Print each number
    Next
End Sub)
t.Start()

```

Visual Prolog

Anonymous functions (in general anonymous *predicates*) were introduced in Visual Prolog in version 7.2.^[14] Anonymous predicates can capture values from the context. If created in an object member it can also access the object state (by capturing This).

`mkAdder` returns an anonymous function, which has captured the argument `X` in the closure. The returned function is a function that adds `X` to its argument:

```

clauses
    mkAdder(X) = { (Y) = X+Y } .

```

References

- [1] ; "JavaScript anonymous functions" (<http://helephant.com/2008/08/javascript-anonymous-functions/>). . Retrieved 2011-02-17.
"Anonymous functions are functions that are dynamically declared at runtime that don't have to be given a name. ... [They] are declared using the function operator. You can use the function operator to create a new function wherever it's valid to put an expression. For example you could declare a new function as a parameter to a function call or to assign a property of another object."
- [2] "Anonymous Functions (C# Programming Guide)" (<http://msdn.microsoft.com/de-de/library/bb882516.aspx>). . Retrieved 2011-02-17.
"There are two kinds of anonymous functions, which are discussed individually in the following topics: Lambda Expressions (C# Programming Guide) Anonymous Methods (C# Programming Guide)"
- [3] https://en.wikibooks.org/wiki/Ruby_Programming/Syntax/Method_Calls#Understanding_blocks.2C_Procs_and_methods
- [4] "Anonymous functions" (<http://www.tu-chemnitz.de/docs/php/functions.anonymous.html>). . Retrieved 2011-02-17. "Anonymous functions, also known as *closures*, allow the creation of functions which have no specified name."
- [5] ; "OpenJDK: Project Lambda" (<http://openjdk.java.net/projects/lambda/>). . Retrieved 2012-09-24. "JSR 335 (Lambda Expressions for the JavaTM Programming Language) aims to support programming in a multicore environment by adding closures and related features to the Java language. [...] The goal of OpenJDK Project Lambda is to host a prototype implementation of JSR 335 suitable for inclusion in JDK 8, the Reference Implementation of Java SE 8."
- [6] "Groovy Documentation" (<http://groovy.codehaus.org/Closures>). . Retrieved 29 May 2012.
- [7] <http://www.microsoft.com/download/en/details.aspx?id=7029>
- [8] "Programming in Lua - More about Functions" (<http://www.lua.org/pil/6.html>). Archived (<http://web.archive.org/web/20080514220940/http://www.lua.org/pil/6.html>) from the original on 14 May 2008. . Retrieved 2008-04-25.
- [9] http://php.net/create_function the top of the page indicates this with "(PHP 4 >= 4.0.1, PHP 5)"
- [10] <http://wiki.php.net/rfc/closures>
- [11] http://wiki.php.net/rfc/closures#zend_internal_perspective
- [12] <http://www.scala-lang.org/node/133>
- [13] apply manual page (<http://www.tcl.tk/man/tcl8.5/TclCmd/apply.htm>) Retrieved 2012-09-06.
- [14] "Anonymous Predicates" (http://wiki.visual-prolog.com/index.php?title=Language_Reference/Terms#Anonymous_Predicates). . in Visual Prolog Language Reference
<http://www.technetfixes.com/2010/03/c-anonymous-functions.html>

External links

- Anonymous Methods - When Should They Be Used? (<http://www.deltics.co.nz/blog/?p=48>) (blog about anonymous function in Delphi)
- C# Lambda Expressions (<http://planetofcoders.com/c-lambda-expressions/>)

Scheme (programming language)

Scheme

	
Paradigm(s)	multi-paradigm: functional, procedural, meta
Appeared in	1975
Designed by	Guy L. Steele and Gerald Jay Sussman
Stable release	R6RS (ratified standard) (2007)
Typing discipline	strong, dynamic
Scope	lexical
Major implementations	Many. See Category:Scheme implementations
Dialects	T
Influenced by	Lisp, ALGOL, MDL
Influenced	Clojure, Common Lisp, Dylan, EuLisp, Haskell, Hop, JavaScript, Kernel, Lua, R, Racket, Ruby
Usual filename extensions	.scm, .ss

Scheme is a functional programming language and one of the two main dialects of the programming language Lisp. Unlike Common Lisp, the other main dialect, Scheme follows a minimalist design philosophy specifying a small standard core with powerful tools for language extension. Its compactness and elegance have made it popular with educators, language designers, programmers, implementors, and hobbyists. The language's diverse appeal is seen as a strong point, though the consequently wide divergence between implementations is seen as one of the language's weak points.^[1]

Scheme was developed at the MIT AI Lab by Guy L. Steele and Gerald Jay Sussman who introduced it to the academic world via a series of memos, now referred to as the Lambda Papers, over the period 1975–1980. The Scheme language is standardized in the official IEEE standard,^[2] and a de facto standard called the *Revisedⁿ Report on the Algorithmic Language Scheme* (RnRS). The most widely implemented standard is R5RS (1998),^[3] and a new standard R6RS^[4] was ratified in 2007.^[5]

Scheme was the first dialect of Lisp to choose lexical scope and the first to require implementations to perform tail-call optimization. It was also one of the first programming languages to support first-class continuations. It had a significant influence on the effort that led to the development of its sister, Common Lisp.^[6]

History

Origin

Scheme started as an attempt to understand Carl Hewitt's Actor model, for which purpose Steele and Sussman wrote a "tiny Lisp interpreter" using Maclisp and then "added mechanisms for creating actors and sending messages."^[7] Scheme was originally called "Schemer", in the tradition of other Lisp-derived languages like Planner or Conniver. The current name resulted from the authors' use of the ITS operating system, which limited filenames to two components of at most six characters each. Currently, "Schemer" is commonly used to refer to a Scheme

programmer.

R6RS

A new language standardization process began at the 2003 Scheme workshop, with the goal of producing an R6RS standard in 2006. This process broke with the earlier RnRS approach of unanimity.

R6RS features a standard module system, allowing a split between the core language and libraries. A number of drafts of the R6RS specification were released, the final version being R5.97RS. A successful vote resulted in the ratification of the new standard, announced on August 28, 2007.^[4]

Currently the newest releases of various Scheme implementations, such as Chez Scheme, Racket, Ikarus, Larceny and Ypsilon, support the R6RS standard. There is a portable reference implementation of the proposed implicitly phased libraries for R6RS, called psyntax, which loads and bootstraps itself properly on various older Scheme implementations.^[8]

R6RS introduces numerous significant changes to the language.^[9] The source code is now specified in Unicode, and a large subset of Unicode characters may now appear in Scheme symbols and identifiers, and there are other minor changes to the lexical rules. Character data is also now specified in Unicode. Many standard procedures have been moved to the new standard libraries, which themselves form a large expansion of the standard, containing procedures and syntactic forms that were formerly not part of the standard. A new module system has been introduced, and systems for exception handling are now standardized. Syntax-rules has been replaced with a more expressive syntactic abstraction facility (syntax-case) which allows the use of all of Scheme at macro expansion time. Compliant implementations are now *required* to support Scheme's full numeric tower, and the semantics of numbers have been expanded, mainly in the direction of support for the IEEE 754 standard for floating point numerical representation.

R7RS

The R6RS standard has caused controversy because it is seen to have departed from the minimalist philosophy.^{[10][11]} In August 2009, the Scheme Steering Committee which oversees the standardization process announced its intention to recommend splitting Scheme into two languages: a large modern programming language for programmers, and a subset of the large version retaining the minimalism praised by educators and casual implementors;^[1] two working groups were created to work on these two new versions of Scheme. The Scheme Reports Process^[12] site has links to the working groups charters, public discussions and issue tracking system.

Distinguishing features

Scheme is primarily a functional programming language. It shares many characteristics with other members of the Lisp programming language family. Scheme's very simple syntax is based on s-expressions, parenthesized lists in which a prefix operator is followed by its arguments. Scheme programs thus consist of sequences of nested lists. Lists are also the main data structure in Scheme, leading to a close equivalence between source code and data formats (homoiconicity). Scheme programs can easily create and evaluate pieces of Scheme code dynamically.

The reliance on lists as data structures is shared by all Lisp dialects. Scheme inherits a rich set of list-processing primitives such as `cons`, `car` and `cdr` from its Lisp ancestors. Scheme uses strictly but dynamically typed variables and supports first-class functions. Thus, functions can be assigned as values to variables or passed as arguments to functions.

This section concentrates mainly on innovative features of the language, including those features that distinguish Scheme from other Lisps. Unless stated otherwise, descriptions of features relate to the R5RS standard.

In examples provided in this section, the notation "====> result" is used to indicate the result of evaluating the expression on the immediately preceding line. This is the same convention used in R5RS.

Fundamental design features

This subsection describes those features of Scheme that have distinguished it from other programming languages from its earliest days. These are the aspects of Scheme that most strongly influence any product of the Scheme language, and they are the aspects that all versions of the Scheme programming language, from 1973 onward, share.

Minimalism

Scheme is a very simple language, much easier to implement than any other language of comparable expressive power.^[13] This ease is attributable to the use of lambda calculus to derive much of the syntax of the language from more primitive forms. For instance of the 23 s-expression-based syntactic constructs defined in the R5RS Scheme standard, 11 are classed as derived or library forms, which can be written as macros involving more fundamental forms, principally lambda. As R5RS says (R5RS sec. 3.1): "The most fundamental of the variable binding constructs is the lambda expression, because all other variable binding constructs can be explained in terms of lambda expressions."^[3]

Fundamental forms: define, lambda, if, quote, unquote, unquote-splicing, quasiquote, define-syntax, let-syntax, letrec-syntax, syntax-rules, set!

Library forms: do, let, let*, letrec, cond, case, and, or, begin, named let, delay

Example: a macro to implement let as an expression using lambda to perform the variable bindings.

```
(define-syntax let
  (syntax-rules ()
    ((let ((var expr) ...) body ...)
     ((lambda (var ...) body ...) expr ...))))
```

Thus using let as defined above a Scheme implementation would rewrite "(let ((a 1) (b 2)) (+ b a))" as "((lambda (a b) (+ b a)) 1 2)", which reduces implementation's task to that of coding procedure instantiations.

In 1998 Sussman and Steele remarked that the minimalism of Scheme was not a conscious design goal, but rather the unintended outcome of the design process. "We were actually trying to build something complicated and discovered, serendipitously, that we had accidentally designed something that met all our goals but was much simpler than we had intended....we realized that the lambda calculus—a small, simple formalism—could serve as the core of a powerful and expressive programming language."^[7]

Lexical scope

Like most modern programming languages and unlike earlier Lisps such as Maclisp or Emacs Lisp, Scheme is lexically scoped: all possible variable bindings in a program unit can be analyzed by reading the text of the program unit without consideration of the contexts in which it may be called.

This contrasts with dynamic scoping which was characteristic of early Lisp dialects, because of the processing costs associated with the primitive textual substitution methods used to implement lexical scoping algorithms in compilers and interpreters of the day. In those Lisps, it was perfectly possible for a reference to a free variable inside a procedure to refer to quite distinct bindings external to the procedure, depending on the context of the call.

The impetus to incorporate what was, in the early 1970s, an unusual scoping model into their new version of Lisp, came from Sussman's studies of ALGOL. He suggested that ALGOL-like lexical scoping mechanisms would help to realise their initial goal of implementing Hewitt's Actor model in Lisp.^[7]

The key insights on how to introduce lexical scoping into a Lisp dialect were popularized in Sussman and Steele's 1975 Lambda Paper, "Scheme: An Interpreter for Extended Lambda Calculus",^[14] where they adopted the concept of the lexical closure, which had been described in an AI Memo in 1970 by Joel Moses, who attributed the idea to Peter J. Landin.^[15]

Lambda calculus

Alonzo Church's mathematical notation, the lambda calculus, has inspired Lisp's use of "lambda" as a keyword for introducing a procedure, as well as influencing the development of functional programming techniques involving the use of higher-order functions in Lisp. But early Lisps were not suitable expressions of the lambda calculus because of their treatment of free variables.^[7]

The introduction of lexical scope resolved the problem by making an equivalence between some forms of lambda notation and their practical expression in a working programming language. Sussman and Steele showed that the new language could be used to elegantly derive all the imperative and declarative semantics of other programming languages including ALGOL and Fortran, and the dynamic scope of other Lisps, by using lambda expressions not as simple procedure instantiations but as "control structures and environment modifiers."^[16] They introduced continuation-passing style along with their first description of Scheme in the first of the Lambda Papers, and in subsequent papers they proceeded to demonstrate the raw power of this practical use of lambda calculus.

Block structure

Scheme inherits its block structure from earlier block structured languages, particularly ALGOL. In Scheme, blocks are implemented by three *binding constructs*: let, let* and letrec. For instance, the following construct creates a block in which a symbol called var is bound to the number 10:

```
(define var "goose")
;; Any reference to var here will be bound to "goose"
(let ((var 10))
  ; statements go here. Any reference to var here will be bound to 10.
)
;; Any reference to var here will be bound to "goose"
```

Blocks can be nested to create arbitrarily complex block structures according to the need of the programmer. The use of block structuring to create local bindings alleviates the risk of namespace collision that can otherwise occur.

One variant of let, let*, permits bindings to refer to variables defined earlier in the same construct, thus:

```
(let* ((var1 10)
       (var2 (+ var1 12)))
  ; But the definition of var1 could not refer to var2
)
```

The other variant, letrec, is designed to enable mutually recursive procedures to be bound to one another.

```
;; Tabulation of Hofstadter's male and female sequences
(letrec ((female (lambda (n)
                     (if (= n 0) 1
                         (- n (male (female (- n 1)))))))
         (male (lambda (n)
                     (if (= n 0) 0
                         (- n (female (male (- n 1)))))))
        (display "i male(i) female(i)" (newline))
        (do ((i 0 (+ i 1)))
            ((> i 8) #f)
            (display i) (display "    ") (display (male i)) (display "
") (display (female i))
            (newline)))
)
```

(See Hofstadter's male and female sequences for the definitions used in this example)

All procedures bound in a single `letrec` may refer to one another by name, as well as to values of variables defined earlier in the same `letrec`, but they may not refer to *values* defined later in the same `letrec`.

A variant of `let`, the "named let" form, has an identifier after the `let` keyword. This binds the `let` variables to the argument of a procedure whose name is the given identifier and whose body is the body of the `let` form. The body may be repeated as desired by calling the procedure. The named `let` is widely used to implement iteration.

Example: a simple counter

```
(let loop ((n 1))
  (if (<= n 10)
    (begin
      (display n) (newline)
      (loop (+ n 1)))))
```

Like any procedure in Scheme the procedure created in the named `let` is a first class object.

Proper tail recursion

Scheme has an iteration construct, `do`, but it is more idiomatic in Scheme to use tail recursion to express iteration. Standard-conforming Scheme implementations are required to optimize tail calls so as to support an unbounded number of active tail calls (R5RS sec. 3.5)^[3]—a property the Scheme report describes as *proper tail recursion*—making it safe for Scheme programmers to write iterative algorithms using recursive structures, which are sometimes more intuitive. Tail recursive procedures and the *named let* form provide support for iteration using tail recursion.

```
;; Tabulation of squares from 0 to 9:
;; Note: loop is simply an arbitrary symbol used as a label. Any symbol
;; will do.
(let loop ((i 0))
  (if (not (= i 10))
    (begin
      (display i) (display " squared = ") (display (* i i)) (newline)
      (loop (+ i 1)))))
```

First-class continuations

Continuations in Scheme are first-class objects. Scheme provides the procedure `call-with-current-continuation` (also known as `call/cc`) to capture the current continuation by packing it up as an escape procedure bound to a formal argument in a procedure provided by the programmer. (R5RS sec. 6.4)^[3] First-class continuations enable the programmer to create non-local control constructs such as iterators, coroutines, and backtracking.

The following example, a traditional programmer's puzzle, shows that Scheme can handle continuations as first-class objects, binding them to variables and passing them as arguments to procedures.

```
(let* ((yin
         ((lambda (cc) (display "@") cc)
          (call-with-current-continuation (lambda (c) c))))
        (yang
         ((lambda (cc) (display "*") cc)
          (call-with-current-continuation (lambda (c) c)))))

        (yin yang))
```

When executed this code displays a counting sequence:

```
"@*@@***@****@*****@*****@*****@*****..."
```

Shared namespace for procedures and variables

In contrast to Common Lisp, all data and procedures in Scheme share a common namespace, whereas in Common Lisp functions and data have separate namespaces making it possible for a function and a variable to have the same name, and requiring special notation for referring to a function as a value. This is sometimes known as the "Lisp-1/Lisp-2" distinction, referring to the unified namespace of Scheme and the separate namespaces of Common Lisp.^[17]

In Scheme, the same primitives that are used to manipulate and bind data can be used to bind procedures. There is no equivalent of Common Lisp's `defun`, `setf` and `#'` primitives.

```
;; Variable bound to a number:
(define f 10)
f
====> 10
;; Mutation (altering the bound value)
(set! f (+ f f 6))
====> 26
;; Assigning a procedure to the same variable:
(set! f (lambda (n) (+ n 12)))
(f 6)
====> 18
;; Assigning the result of an expression to the same variable:
(set! f (f 1))
f
====> 13
;; functional programming:
(apply + '(1 2 3 4 5 6))
====> 21
(set! f (lambda (n) (+ n 100)))
(map f '(1 2 3))
====> (101 102 103)
```

Implementation standards

This subsection documents design decisions that have been taken over the years which have given Scheme a particular character, but are not the direct outcomes of the original design.

Numerical tower

Scheme specifies a comparatively full set of numerical datatypes including complex and rational types, which is known in Scheme as the numerical tower (R5RS sec. 6.2^[3]). The standard treats these as abstractions, and does not commit the implementor to any particular internal representations.

Numbers may have the quality of exactness. An exact number can only be produced by a sequence of exact operations involving other exact numbers—inequality is thus contagious. The standard specifies that any two implementations must produce equivalent results for all operations resulting in exact numbers.

The R5RS standard specifies procedures `exact->inexact` and `inexact->exact` which can be used to change the exactness of a number. `inexact->exact` produces "the exact number that is numerically closest to

the argument." `exact->inexact` produces "the inexact number that is numerically closest to the argument". The R6RS standard omits these procedures from the main report, but specifies them as R5RS compatibility procedures in the standard library (rnrs r5rs (6)).

In the R5RS standard, Scheme implementations are not required to implement the whole numerical tower, but they must implement "a coherent subset consistent with both the purposes of the implementation and the spirit of the Scheme language" (R5RS sec. 6.2.3).^[3] The new R6RS standard does require implementation of the whole tower, and "exact integer objects and exact rational number objects of practically unlimited size and precision, and to implement certain procedures...so they always return exact results when given exact arguments" (R6RS sec. 3.4, sec. 11.7.1).^[4]

Example 1: exact arithmetic in an implementation that supports exact rational complex numbers.

```
;; Sum of three rational real numbers and two rational complex numbers
(define x (+ 1/3 1/4 -1/5 -1/3i 405/50+2/3i))
x
====> 509/60+1/3i
;; Check for exactness.
(exact? x)
====> #t
```

Example 2: Same arithmetic in an implementation that supports neither exact rational numbers nor complex numbers but does accept real numbers in rational notation.

```
;; Sum of four rational real numbers
(define xr (+ 1/3 1/4 -1/5 405/50))
;; Sum of two rational real numbers
(define xi (+ -1/3 2/3))
xr
====> 8.48333333333333
xi
====> 0.33333333333333
;; Check for exactness.
(exact? xr)
====> #f
(exact? xi)
====> #f
```

Both implementations conform to the R5RS standard but the second does not conform to R6RS because it does not implement the full numerical tower.

Delayed evaluation

Scheme supports delayed evaluation through the `delay` form and the procedure `force`.

```
(define a 10)
(define eval-aplus2 (delay (+ a 2)))
(set! a 20)
(force eval-aplus2)
====> 22
(define eval-aplus50 (delay (+ a 50)))
(let ((a 8))
```

```
(force eval-aplus50))
====> 70
(set! a 100)
(force eval-aplus2)
====> 22
```

The lexical context of the original definition of the promise is preserved, and its value is also preserved after the first use of `force`. The promise is only ever evaluated once.

These primitives, which produce or handle values known as promises, can be used to implement advanced lazy evaluation constructs such as streams.^[18]

In the R6RS standard, these are no longer primitives, but instead are provided as part of the R5RS compatibility library (rnrs r5rs (6)).

In R5RS, a suggested implementation of `delay` and `force` is given, implementing the promise as a procedure with no arguments (a thunk) and using memoization to ensure that it is only ever evaluated once, irrespective of the number of times `force` is called (R5RS sec. 6.4).^[3]

SRFI 41 enables the expression of both finite and infinite sequences with extraordinary economy. For example this is a definition of the fibonacci sequence using the functions defined in SRFI 41.^[18]

```
;; Define the Fibonacci sequence:
(define fibs
  (stream-cons 0
    (stream-cons 1
      (stream-map +
        fibs
        (stream-cdr fibs)))))

;; Compute the hundredth number in the sequence:
(stream-ref fibs 99)
====> 218922995834555169026
```

Order of evaluation of procedure arguments

Most Lisps specify an order of evaluation for procedure arguments. Scheme does not. Order of evaluation—including the order in which the expression in the operator position is evaluated—may be chosen by an implementation on a call-by-call basis, and the only constraint is that "the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation." (R5RS sec. 4.1.3)^[3]

```
(let ((ev (lambda (n) (display "Evaluating ")
                           (display (if (procedure? n) "procedure" n))
                           (newline) n)))
      ((ev +) (ev 1) (ev 2)))
  ===> 3
```

`ev` is a procedure that describes the argument passed to it, then returns the value of the argument. In contrast with other Lisps, the appearance of an expression in the operator position (the first item) of a Scheme expression is quite legal, as long as the result of the expression in the operator position is a procedure.

In calling the procedure "+" to add 1 and 2, the expressions `(ev +)`, `(ev 1)` and `(ev 2)` may be evaluated in any order, as long as the effect is not as if they were evaluated in parallel. Thus the following three lines may be displayed in any order by standard Scheme when the above example code is executed, although the text of one line may not be

interleaved with another, because that would violate the sequential evaluation constraint.

Evaluating 1

Evaluating 2

Evaluating procedure

Hygienic macros

In the R5RS standard and also in later reports, the syntax of Scheme can easily be extended via the macro system. The R5RS standard introduced a powerful hygienic macro system that allows the programmer to add new syntactic constructs to the language using a simple pattern matching sublanguage (R5RS sec 4.3).^[3] Prior to this, the hygienic macro system had been relegated to an appendix of the R4RS standard, as a "high level" system alongside a "low level" macro system, both of which were treated as extensions to Scheme rather than an essential part of the language.^[19]

Implementations of the hygienic macro system, also called `syntax-rules`, are required to respect the lexical scoping of the rest of the language. This is assured by special naming and scoping rules for macro expansion, and avoids common programming errors that can occur in the macro systems of other programming languages. R6RS specifies a more sophisticated transformation system, `syntax-case`, which has been available as a language extension to R5RS Scheme for some time.

```
;; Define a macro to implement a variant of "if" with a multi-expression
;; true branch and no false branch.

(define-syntax when
  (syntax-rules ()
    ((when pred exp exps ...)
     (if pred (begin exp exps ...)))))
```

Invocations of macros and procedures bear a close resemblance—both are s-expressions—but they are treated differently. When the compiler encounters an s-expression in the program, it first checks to see if the symbol is defined as a syntactic keyword within the current lexical scope. If so, it then attempts to expand the macro, treating the items in the tail of the s-expression as arguments without compiling code to evaluate them, and this process is repeated recursively until no macro invocations remain. If it is not a syntactic keyword, the compiler compiles code to evaluate the arguments in the tail of the s-expression and then to evaluate the variable represented by the symbol at the head of the s-expression and call it as a procedure with the evaluated tail expressions passed as actual arguments to it.

Most Scheme implementations also provide additional macro systems. Among popular ones are syntactic closures, explicit renaming macros and `define-macro`, a non-hygienic macro system similar to `defmacro` system provided in Common Lisp.

Environments and eval

Prior to R5RS, Scheme had no standard equivalent of the `eval` procedure which is ubiquitous in other Lisps, although the first Lambda Paper had described `evaluate` as "similar to the LISP function EVAL"^[14] and the first Revised Report in 1978 replaced this with `enclose`, which took two arguments. The second, third and fourth revised reports omitted any equivalent of `eval`.

The reason for this confusion is that in Scheme with its lexical scoping the result of evaluating an expression depends on where it is evaluated. For instance, it is not clear whether the result of evaluating the following expression should be 5 or 6:^[20]

```
(let ((name '+))
  (let ((+ *))
    ...))
```

```
(evaluate (list name 2 3)))
```

If it is evaluated in the outer environment, where `name` is defined, the result is the sum of the operands. If it is evaluated in the inner environment, where the symbol "+" has been bound to the value of the procedure "*", the result is the product of the two operands.

R5RS resolves this confusion by specifying three procedures that return environments, and providing a procedure `eval` that takes an s-expression and an environment and evaluates the expression in the environment provided. (R5RS sec. 6.5)^[3] R6RS extends this by providing a procedure called `environment` by which the programmer can specify exactly which objects to import into the evaluation environment.

Treatment of non-boolean values in boolean expressions

In most dialects of Lisp including Common Lisp, by convention the value `NIL` evaluates to the value false in a boolean expression. In Scheme, since the IEEE standard in 1991,^[2] all values except `#f`, including `NIL`'s equivalent in Scheme which is written as `'()`, evaluate to the value true in a boolean expression. (R5RS sec. 6.3.1)^[3]

Where the constant representing the boolean value of true is `T` in most Lisps, in Scheme it is `#t`.

Disjointness of primitive datatypes

In Scheme the primitive datatypes are disjoint. Only one of the following predicates can be true of any Scheme object: `boolean?`, `pair?`, `symbol?`, `number?`, `char?`, `string?`, `vector?`, `port?`, `procedure?`. (R5RS sec 3.2)^[3]

Within the numerical datatype, by contrast, the numerical values overlap. For example, an integer value satisfies all of the `integer?`, `rational?`, `real?`, `complex?` and `number?` predicates at the same time. (R5RS sec 6.2)^[3]

Equivalence predicates

Scheme has three different types of equivalence between arbitrary objects denoted by three different *equivalence predicates*, relational operators for testing equality, `eq?`, `eqv?` and `equal?`:

- `eq?` evaluates to `#f` unless its parameters represent the same data object in memory;
- `eqv?` is generally the same as `eq?` but treats primitive objects (e.g. characters and numbers) specially so that numbers that represent the same value are `eqv?` even if they do not refer to the same object;
- `equal?` compares data structures such as lists, vectors and strings to determine if they have congruent structure and `eqv?` contents.(R5RS sec. 6.1)^[3]

Type dependent equivalence operations also exist in Scheme: `string=?` and `string-ci=?` compare two strings (the latter performs a case-independent comparison); `char=?` and `char-ci=?` compare characters; `=` compares numbers.^[3]

Comments

Up to the R5RS standard, the standard comment in Scheme was a semicolon, which makes the rest of the line invisible to Scheme. Numerous implementations have supported alternative conventions permitting comments to extend for more than a single line, and the R6RS standard permits two of them: an entire s-expression may be turned into a comment (or "commented out") by preceding it with `#;` (introduced in SRFI 62^[21]) and a multiline comment or "block comment" may be produced by surrounding text by `#|` and `|#`.

Input/output

Scheme's input and output is based on the *port* datatype. (R5RS sec 6.6)^[3] R5RS defines two default ports, accessible with the procedures `current-input-port` and `current-output-port`, which correspond to the Unix notions of standard input and standard output. Most implementations also provide `current-error-port`. Redirection of input and standard output is supported in the standard, by standard procedures such as `with-input-from-file` and `with-output-to-file`. Most implementations provide string ports with similar redirection capabilities, enabling many normal input-output operations to be performed on string buffers instead of files, using procedures described in SRFI 6.^[22] The R6RS standard specifies much more sophisticated and capable port procedures and many new types of port.

The following examples are written in strict R5RS Scheme.

Example 1: With output defaulting to (`current-output-port`):

```
(let ((hello0 (lambda () (display "Hello world") (newline))))
  (hello0))
```

Example 2: As 1, but using optional port argument to output procedures

```
(let ((hello1 (lambda (p) (display "Hello world" p) (newline p))))
  (hello1 (current-output-port)))
```

Example 3: As 1, but output is redirected to a newly created file

```
;; NB: with-output-to-file is an optional procedure in R5RS
(let ((hello0 (lambda () (display "Hello world") (newline))))
  (with-output-to-file "helloworldoutputfile" hello0))
```

Example 4: As 2, but with explicit file open and port close to send output to file

```
(let ((hello1 (lambda (p) (display "Hello world" p) (newline p)))
      (output-port (open-output-file "helloworldoutputfile")))
  (hello1 output-port)
  (close-output-port output-port))
```

Example 5: As 2, but with using call-with-output-file to send output to a file.

```
(let ((hello1 (lambda (p) (display "Hello world" p) (newline p))))
  (call-with-output-file "helloworldoutputfile" hello1))
```

Similar procedures are provided for input. R5RS Scheme provides the predicates `input-port?` and `output-port?`. For character input and output, `write-char`, `read-char`, `peek-char` and `char-ready?` are provided. For writing and reading Scheme expressions, Scheme provides `read` and `write`. On a read operation, the result returned is the `end-of-file` object if the input port has reached the end of the file, and this can be tested using the predicate `eof-object?`.

In addition to the standard, SRFI 28 defines a basic formatting procedure resembling Common Lisp's `format` function, after which it is named.^[23]

Redefinition of standard procedures

In Scheme, procedures are bound to variables. At R5RS the language standard formally mandated that programs may change the variable bindings of built-in procedures, effectively redefining them. (R5RS "Language changes")^[3] For example, one may extend `+` to accept strings as well as numbers by redefining it:

```
(set! +
  (let ((original+ +))
    (lambda args
      (if (and (not (null? args)) (string? (car args)))
          (apply string-append args)
          (apply original+ args)))))

(+ 1 2 3)
====> 6
(+ "1" "2" "3")
====> "123"
```

In R6RS every binding, including the standard ones, belongs to some library, and all exported bindings are immutable. (R6RS sec 7.1)^[4] Because of this, redefinition of standard procedures by mutation is forbidden. Instead, it is possible to import a different procedure under the name of a standard one, which in effect is similar to redefinition.

Nomenclature and naming conventions

In Standard Scheme, procedures that convert from one datatype to another contain the character string "`->`" in their name, predicates end with a "?", and procedures that change the value of already-allocated data end with a "!". These conventions are often followed by Scheme programmers.

In formal contexts such as Scheme standards, the word "procedure" is used in preference to "function" to refer to a lambda expression or primitive procedure. In normal usage the words "procedure" and "function" are used interchangeably. Procedure application is sometimes referred to formally as *combination*.

As in other Lisps, the term "thunk" is used in Scheme to refer to a procedure with no arguments. The term "proper tail recursion" refers to the property of all Scheme implementations, that they perform tail-call optimization so as to support an indefinite number of active tail calls.

The form of the titles of the standards documents since R3RS, "Revisedⁿ Report on the Algorithmic Language Scheme", is a reference to the title of the ALGOL 60 standard document, "Revised Report on the Algorithmic Language Algol 60." The Summary page of R3RS is closely modeled on the Summary page of the ALGOL 60 Report.^{[24][25]}

Review of standard forms and procedures

The language is formally defined in the standards R5RS (1998) and R6RS (2007). They describe standard "forms": keywords and accompanying syntax, which provide the control structure of the language, and standard procedures which perform common tasks.

Standard forms

This table describes the standard forms in Scheme. Some forms appear in more than one row because they cannot easily be classified into a single function in the language.

Forms marked "L" in this table are classed as derived "library" forms in the standard and are often implemented as macros using more fundamental forms in practice, making the task of implementation much easier than in other

languages.

Standard forms in the language R5RS Scheme

Purpose	Forms
Definition	define
Binding constructs	lambda, do (L), let (L), let* (L), letrec (L)
Conditional evaluation	if, cond (L), case (L), and (L), or (L)
Sequential evaluation	begin (*)
Iteration	lambda, do (L), named let (L)
Syntactic extension	define-syntax, let-syntax, letrec-syntax, syntax-rules (R5RS), syntax-case (R6RS)
Quoting	quote('), unquote(,), quasiquote(`), unquote-splicing(@)
Assignment	set!
Delayed evaluation	delay (L)

Note that `begin` is defined as a library syntax in R5RS, but the expander needs to know about it to achieve the splicing functionality. In R6RS it is no longer a library syntax.

Standard procedures

The following two tables describe the standard procedures in R5RS Scheme. R6RS is far more extensive and a summary of this type would not be practical.

Some procedures appear in more than one row because they cannot easily be classified into a single function in the language.

Standard procedures in the language R5RS Scheme

Purpose	Procedures
Construction	vector, make-vector, make-string, list
Equivalence predicates	eq?, eqv?, equal?, string=?, string-ci=?, char=?, char-ci=?
Type conversion	vector->list, list->vector, number->string, string->number, symbol->string, string->symbol, char->integer, integer->char, string->list, list->string
Numbers	<i>See separate table</i>
Strings	string?, make-string, string, string-length, string-ref, string-set!, string=?, string-ci=?, string<? string-ci<?, string<=? string-ci<?, string>? string-ci>?, string>? string-ci>?, substring, string-append, string->list, list->string, string-copy, string-fill!
Characters	char?, char=?, char-ci=?, char<? char-ci<?, char<=? char-ci<?, char>? char-ci>?, char>=? char-ci>?, char-alphabetic?, char-numeric?, char-whitespace?, char-upper-case?, char-lower-case?, char->integer, integer->char, char-upcase, char-downcase
Vectors	make-vector, vector, vector?, vector-length, vector-ref, vector-set!, vector->list, list->vector, vector-fill!
Symbols	symbol->string, string->symbol, symbol?
Pairs and lists	pair?, cons, car, cdr, set-car!, set-cdr!, null?, list?, list, length, append, reverse, list-tail, list-ref, memq, memv, member, assq, assv, assoc, list->vector, vector->list, list->string, string->list
Identity predicates	boolean?, pair?, symbol?, number?, char?, string?, vector?, port?, procedure?
Continuations	call-with-current-continuation (call/cc), values, call-with-values, dynamic-wind

Environments	eval, scheme-report-environment, null-environment, interaction-environment (optional)
Input/output	display, newline, read, write, read-char, write-char, peek-char, char-ready?, eof-object? open-input-file, open-output-file, close-input-port, close-output-port, input-port?, output-port?, current-input-port, current-output-port, call-with-input-file, call-with-output-file, with-input-from-file(optional), with-output-to-file(optional)
System interface	load (optional), transcript-on (optional), transcript-off (optional)
Functional programming	procedure?, apply, map, for-each
Booleans	boolean? not

String and character procedures that contain "-ci" in their names perform case-independent comparisons between their arguments: upper case and lower case versions of the same character are taken to be equal.

Standard numeric procedures in the language R5RS Scheme

Purpose	Procedures
Basic arithmetic operators	+, -, *, /, abs, quotient, remainder, modulo, gcd, lcm, expt, sqrt
Rational numbers	numerator, denominator, rational?, rationalize
Approximation	floor, ceiling, truncate, round
Exactness	inexact->exact, exact->inexact, exact?, inexact?
Inequalities	<, <=, >, >=, =
Miscellaneous predicates	zero?, negative?, positive? odd? even?
Maximum and minimum	max, min
Trigonometry	sin, cos, tan, asin, acos, atan
Exponentials	exp, log
Complex numbers	make-rectangular, make-polar, real-part, imag-part, magnitude, angle, complex?
Input-output	number->string, string->number
Type predicates	integer?, rational?, real?, complex?, number?

Implementations of - and / that take more than two arguments are defined but left optional at R5RS.

Scheme Requests for Implementation

Because of Scheme's minimalism, many common procedures and syntactic forms are not defined by the standard. In order to keep the core language small but facilitate standardization of extensions, the Scheme community has a "Scheme Request for Implementation" (SRFI) process by which extension libraries are defined through careful discussion of extension proposals. This promotes code portability. Many of the SRFIs are supported by all or most Scheme implementations.

SRFIs with fairly wide support in different implementations include:^[26]

- 0: feature-based conditional expansion construct
- 1: list library
- 4: homogeneous numeric vector datatypes
- 6: basic string ports
- 8: receive, binding to multiple values
- 9: defining record types
- 13: string library
- 14: character-set library

- 16: syntax for procedures of variable arity
- 17: generalized set!
- 18: Multithreading support
- 19: time data types and procedures
- 25: multi-dimensional array primitives
- 26: notation for specializing parameters without currying
- 27: sources of random bits
- 28: basic format strings
- 29: localization
- 30: nested multi-line comments
- 31: a special form for recursive evaluation
- 37: args-fold: a program argument processor
- 39: parameter objects
- 41: streams
- 42: eager comprehensions
- 43: vector library
- 45: primitives for expressing iterative lazy algorithms
- 60: integers as bits
- 61: a more general cond clause
- 66: octet vectors
- 67: compare procedures

A full list of accepted (finalized) SRFIs is available at <http://srfi.schemers.org/final-srfis.html>

Implementations

Main category: Scheme implementations

The elegant, minimalist design has made Scheme a popular target for language designers, hobbyists, and educators, and because of the small size of a typical interpreter it is also a popular choice for embedded systems and scripting. This has resulted in scores of implementations,^[27] most of which differ from each other so much that porting programs from one implementation to another is quite difficult; and the small size of the standard language means that writing a useful program of any great complexity in standard, portable Scheme is almost impossible. The R6RS standard specifies a much broader language, in an attempt to broaden its appeal to programmers.

Almost all implementations provide a traditional Lisp-style read–eval–print loop for development and debugging. Many also compile Scheme programs to executable binary. Support for embedding Scheme code in programs written in other languages is also common, as the relative simplicity of Scheme implementations makes Scheme a popular choice for adding scripting capabilities to larger systems developed in languages such as C. Gambit, Chicken and Bigloo work by compiling Scheme to C, which makes embedding particularly easy. In addition, Bigloo's compiler can be configured to generate JVM bytecode, and it also features an experimental bytecode generator for .Net.

Some implementations support additional features. For example, Kawa and JScheme provide integration with Java classes, and the Scheme to C compilers often make it easy to use external libraries written in C, up to allowing the embedding of actual C code in the Scheme source. Another example is Pvts, which offers a set of visual tools for supporting the learning of Scheme.

Usage

Scheme is widely used by a number^[28] of schools; in particular, a number of introductory Computer Science courses use Scheme in conjunction with the textbook *Structure and Interpretation of Computer Programs* (SICP).^[29] For the past 12 years, PLT has run the ProgramByDesign (formerly TeachScheme!) project, which has exposed close to 600 high school teachers and thousands of high school students to rudimentary Scheme programming. MIT's old introductory programming class 6.001 was taught in Scheme.^[30] Although 6.001 has been replaced by more modern courses, SICP continues to be taught at MIT.^[31] The textbook *How to Design Programs* by Matthias Felleisen, currently at Northeastern University, is used by some institutes of higher education for their introductory computer science courses. Both Northeastern University and Worcester Polytechnic Institute use Scheme exclusively for their introductory courses Fundamentals of Computer Science (CS2500) and Introduction to Program Design (CS1101), respectively.^{[32][33]} Indiana University's introductory class, C211, is taught entirely in Scheme. The introductory class at UC Berkeley, CS 61A, was until recently taught entirely in Scheme, save minor diversions into Logo to demonstrate dynamic scope; all course materials, including lecture webcasts, are available online free of charge.^[34] The introductory computer science courses at Yale and Grinnell College are also taught in Scheme.^[35] Several introductory Computer Science courses at Rice University are also taught in Scheme.^[36] Programming Design Paradigms,^[32] a mandatory course for the Computer science Graduate Students at Northeastern University, also extensively uses Scheme. The introduction Computer Science course at the University of Minnesota - Twin Cities, CSci 1901, also uses Scheme as its primary language, followed by a course that introduces students to the Java programming language.^[37] LambdaBeans^[38] is an open source Scheme editor, helping with syntax coloring, code completion, and other features typical to code editors. In the software industry, Tata Consultancy Services, Asia's largest software consultancy firm, uses Scheme in their month-long training program for fresh college graduates. Although there are relatively few examples of Scheme in apparent usage^[39] for non-pedagogical purposes, it is/was used for the following:

- Monk^[40], an implementation developed by SeeBeyond to support extending application functionality in their enterprise application integration tools.
- The Document Style Semantics and Specification Language (DSSSL), which provides a method of specifying SGML stylesheets, uses a Scheme subset.^[41]
- The well-known open source raster graphics editor GIMP uses Scheme as a scripting language.^[42]
- Guile has been adopted by GNU project as its official scripting language, and that implementation of Scheme is embedded in such applications as GNU LilyPond and GnuCash as a scripting language for extensions. Likewise, Guile used to be the scripting language for the desktop environment GNOME,^[43] and GNOME still has a project that provides Guile bindings to its library stack.^[44]
- Elk Scheme is used by Synopsys as a scripting language for its technology CAD (TCAD) tools.^[45]
- Shiro Kawai, senior programmer on the movie *Final Fantasy: The Spirits Within*, used Scheme as a scripting language for managing the real-time rendering engine.^[46]
- Google App Inventor for Android uses Scheme, where Kawa is used to compile the Scheme code down to byte-codes for the Java Virtual Machine running on Android devices.^[47]

References

- [1] Will Clinger, Marc Feeley, Chris Hanson, Jonathan Rees and Olin Shivers (2009-08-20). "Position Statement, *draft*" (<http://scheme-reports.org/2009/position-statement.html>). Scheme Steering Committee. . Retrieved 2012-08-09.
- [2] 1178-1990 (Reaff 2008) IEEE Standard for the Scheme Programming Language. IEEE part number STDPD14209, unanimously reaffirmed (<http://standards.ieee.org/board/rev/308minutes.html>) at a meeting of the IEEE-SA Standards Board Standards Review Committee (RevCom), March 26, 2008 (item 6.3 on minutes), reaffirmation minutes accessed October 2009. NOTE: this document is only available for purchase from IEEE and is not available online at the time of writing (2009).
- [3] Richard Kelsey, William Clinger, Jonathan Rees et al. (August 1998). "Revised⁵ Report on the Algorithmic Language Scheme" (<http://www.schemers.org/Documents/Standards/R5RS/>). *Higher-Order and Symbolic Computation* 11 (1): 7–105. doi:10.1023/A:1010051815785. . Retrieved 2012-08-09.
- [4] Michael Sperber, R. Kent Dybvig, Matthew Flatt, Anton Van Straaten et al. (August 2007). "Revised⁶ Report on the Algorithmic Language Scheme (R6RS)" (<http://www.r6rs.org/>). Scheme Steering Committee. . Retrieved 2012-08-09.
- [5] "R6RS ratification-voting results" (<http://www.r6rs.org/ratification/results.html>). Scheme Steering Committee. 2007-08-13. . Retrieved 2012-08-09.
- [6] Common LISP: The Language, 2nd Ed., Guy L. Steele Jr. Digital Press; 1981. ISBN 978-1-55558-041-4. "Common Lisp is a new dialect of Lisp, a successor to MacLisp, influenced strongly by ZetaLisp and to some extent by Scheme and InterLisp."
- [7] Gerald Jay Sussman and Guy L. Steele, Jr. (December 1998). "The First Report on Scheme Revisited" (<http://www.brics.dk/~hosc/local/HOSC-11-4-pp399-404.pdf>) (PDF). *Higher-Order and Symbolic Computation* 11 (4): 399–404. doi:10.1023/A:1010079421970. ISSN 1388-3690. . Retrieved 2012-08-09.
- [8] Abdulaziz Ghuloum (2007-10-27). "R6RS Libraries and syntax-case system (psyntax)" (<https://www.cs.indiana.edu/~aghuloum/r6rs-libraries/>). Ikarus Scheme. . Retrieved 2009-10-20.
- [9] "Revised⁶ Report on the Algorithmic Language Scheme, Appendix E: language changes" (http://www.r6rs.org/final/html/r6rs-r6rs-Z-H-19.html#node_chap_E). Scheme Steering Committee. 2007-09-26. . Retrieved 2009-10-20.
- [10] "R6RS Electorate" (<http://www.r6rs.org/ratification/electorate.html>). Scheme Steering Committee. 2007. . Retrieved 2012-08-09.
- [11] Marc Feeley (compilation) (2007-10-26). "Implementors' intentions concerning R6RS" (<http://lists.r6rs.org/pipermail/r6rs-discuss/2007-October/003351.html>). Scheme Steering Committee, r6rs-discuss mailing list. . Retrieved 2012-08-09.
- [12] <http://www.scheme-reports.org/>
- [13] Indeed, the Scheme 48 implementation is so-named because the interpreter was written by Richard Kelsey and Jonathan Rees in 48 hours on August 6th and 7th, 1986. See Richard Kelsey, Jonathan Rees, Mike Sperber (2008-01-10). "The Incomplete Scheme 48 Reference Manual for release 1.8" (<http://s48.org/1.8/manual/manual.html>). Jonathan Rees, s48.org. . Retrieved 2012-08-09.
- [14] Gerald Jay Sussman and Guy Lewis Steele, Jr. (December 1975). "Scheme: An Interpreter for Extended Lambda Calculus" (<http://library.readscheme.org/page1.html>) (postscript or PDF). *AI Memos* (MIT AI Lab) **AIM-349**. . Retrieved 2012-08-09
- [15] Joel Moses (June 1970) (PDF), *The Function of FUNCTION in LISP, or Why the FUNARG Problem Should Be Called the Environment Problem* (<http://dspace.mit.edu/handle/1721.1/5854>), AI Memo 199, , retrieved 2012-08-09, "A useful metaphor for the difference between FUNCTION and QUOTE in LISP is to think of QUOTE as a porous or an open covering of the function since free variables escape to the current environment. FUNCTION acts as a closed or nonporous covering (hence the term "closure" used by Landin). Thus we talk of "open" Lambda expressions (functions in LISP are usually Lambda expressions) and "closed" Lambda expressions. [...] My interest in the environment problem began while Landin, who had a deep understanding of the problem, visited MIT during 1966-67. I then realized the correspondence between the FUNARG lists which are the results of the evaluation of "closed" Lambda expressions in LISP and ISWIM's Lambda Closures."
- [16] Gerald Jay Sussman and Guy Lewis Steele, Jr. (March 1976). "Lambda: The Ultimate Imperative" (<http://library.readscheme.org/page1.html>) (postscript or PDF). *AI Memos* (MIT AI Lab) **AIM-353**. . Retrieved 2012-08-09
- [17] Gabriel, Richard P.; Pitman, Kent (1988). "Technical Issues of Separation in Function Cells and Value Cells" (<http://www.nhplace.com/kent/Papers/Technical-Issues.html>). *Lisp and Symbolic Computation* 1 (1): pp. 81–101. June 1988. doi:10.1007/BF01806178. . Retrieved 2012-08-09
- [18] Philip L. Bewig (2008-01-24). "SRFI 41: Streams" (<http://srfi.schemers.org/srfi-41/srfi-41.html>). The SRFI Editors, schemers.org. . Retrieved 2012-08-09.
- [19] William Clinger and Jonathan Rees, editors (1991). "Revised⁴ Report on the Algorithmic Language Scheme" (http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_toc.html). *ACM Lisp Pointers* 4 (3): 1–55. . Retrieved 2012-08-09
- [20] Jonathan Rees, The Scheme of Things The June 1992 Meeting (<http://mumble.net/~jar/pubs/scheme-of-things/june-92-meeting.ps>) (postscript), in *Lisp Pointers*, V(4), October–December 1992. Retrieved 2012-08-09
- [21] Taylor Campbell (2005-07-21). "SRFI 62: S-expression comments" (<http://srfi.schemers.org/srfi-62/srfi-62.html>). The SRFI Editors, schemers.org. . Retrieved 2012-08-09.
- [22] William D Clinger (1999-07-01). "SRFI 6: Basic String Ports" (<http://srfi.schemers.org/srfi-6/srfi-6.html>). The SRFI Editors, schemers.org. . Retrieved 2012-08-09.
- [23] Scott G. Miller (2002-06-25). "SRFI 28: Basic Format Strings" (<http://srfi.schemers.org/srfi-28/srfi-28.html>). The SRFI Editors, schemers.org. . Retrieved 2012-08-09.

- [24] J.W. Backus, F.L. Bauer, J.Green, C. Katz, J. McCarthy P. Naur et al. (January–April, 1960). "Revised Report on the Algorithmic Language Algol 60" (<http://www.masswerk.at/algol60/report.htm>). *Numerische Mathematik, Communications of the ACM, and Journal of the British Computer Society*. . Retrieved 2012-08-09
- [25] Jonathan Rees and William Clinger (Editors) (December 1986). "Revised(3) Report on the Algorithmic Language Scheme, (Dedicated to the Memory of ALGOL 60)" (http://groups.csail.mit.edu/mac/ftpdir/scheme-reports/r3rs-html/r3rs_toc.html). *ACM SIGPLAN Notices* **21** (12): 37–79. . Retrieved 2012-08-09
- [26] "Scheme Systems Supporting SRFIs" (<http://srfi.schemers.org/srfi-implementers.html>). The SRFI Editors, schemers.org. 2009-08-30. . Retrieved 2012-08-09.
- [27] 75 known implementations of Scheme are listed by "scheme-faq-standards" (<http://community.schemewiki.org/?scheme-faq-standards#implementations>). Community Scheme Wiki. 2009-06-25. . Retrieved 2009-10-20.
- [28] Ed Martin (2009-07-20). "List of Scheme-using schools" (<http://www.schemers.com/schools.html>). Schemers Inc.. . Retrieved 2009-10-20.
- [29] "List of SICP-using schools" (<http://mitpress.mit.edu/sicp/adopt-list.html>). MIT Press. 1999-01-26. . Retrieved 2009-10-20.
- [30] Eric Grimson (Spring, 2005). "6.001 Structure and Interpretation of Computer Programs" (<http://ocw.mit.edu/OcwWeb/Electrical-Engineering-and-Computer-Science/6-001Spring-2005/CourseHome/index.htm>). MIT Open Courseware. . Retrieved 2009-10-20.
- [31] Alex Vandiver, Nelson Elhage, et al (January 2009). "6.184 - Zombies drink caffeinated 6.001" (<http://web.mit.edu/alexmv/6.001/>). MIT CSAIL. . Retrieved 2009-10-20.
- [32] CS 2500: Fundamentals of Computer Science I (<http://www.ccs.neu.edu/course/cs2500/>), Northeastern University
- [33] CS 1101: Introduction to Program Design (A05): course software (<http://web.cs.wpi.edu/~cs1101/a05/details.html#software>), Worcester Polytechnic Institute
- [34] Brian Harvey (Fall 2009). "Computer Science 61A, Berkeley" (<http://inst.eecs.berkeley.edu/~cs61a/>). Department of Electrical Engineering and Computer Sciences, Berkeley. . Retrieved 2009-10-20.
- [35] Dana Angluin (Fall 2009). "Introduction to Computer Science (CPSC 201)" (<http://zoo.cs.yale.edu/classes/cs201/>). The Zoo, Yale University Computer Science Department. . Retrieved 2009-10-20.
- [36] "Computer Science Courses 100-400, Rice University" (http://cohesion.rice.edu/engineering/computerscience/about.cfm?doc_id=8045). Rice University Computer Science Department. Fall 2009. . Retrieved 2009-10-20.
- [37] University of Minnesota Computer Science Department <http://www-users.itlabs.umn.edu/classes/Spring-2010/csci1901/Structure of Computer Programming I>, University of Minnesota Spring 2010 | accessdate=2010-1-30
- [38] Geertjan Wieleng (2009-04-13). "Scheme Editor on the NetBeans Platform" (<http://netbeans.dzone.com/news/scheme-editor-netbeans>). DZone, Inc.. . Retrieved 2012-08-09.
- [39] See *What is Scheme used for?* at "scheme-faq-general" (<http://community.schemewiki.org/?scheme-faq-general>). Community Scheme Wiki. 2009-03-21. . Retrieved 2012-08-09.
- [40] http://download.oracle.com/docs/cd/E18867_01/4.5.x/4.5.3/eGate_Integrator/Monk_Reference.pdf
- [41] Robin Cover (2002-02-25). "DSSSL - Document Style Semantics and Specification Language. ISO/IEC 10179:1996" (<http://xml.coverpages.org/dsss.html>). Cover Pages. . Retrieved 2012-08-09.
- [42] "The major scripting language for the GIMP that has been attached to it today is Scheme." From Dov Grobgeld (2002). "The GIMP Basic Scheme Tutorial" (http://www.gimp.org/tutorials/Basic_Scheme/). The GIMP Team. . Retrieved 2012-08-09.
- [43] Todd Graham Lewis, David Zoll, Julian Missig (2002). "GNOME FAQ from [[Internet Archive (<http://web.archive.org/web/20000522010523/http://www.gnome.org/gnomefaq/html/x930.html>)]]". The Gnome Team, gnome.org. Archived from the original (<http://www.gnome.org/gnomefaq/html/x930.html>) on 2000-05-22. . Retrieved 2012-08-09.
- [44] "guile-gnome" (<http://www.gnu.org/software/guile-gnome/>). Free Software Foundation. . Retrieved 2012-08-09.
- [45] Laurence Brevard (2006-11-09). "Synopsys MAP-inSM Program Update: EDA Interoperability Developers' Forum" (http://www.synopsys.com/community/interoperability/documents/devforum_pres/2006nov/milkywaysession_mapin_overview.pdf) (PDF). Synopsis Inc. . Retrieved 2012-08-09.
- [46] Kawai, Shiro (October 2002). "Gluing Things Together - Scheme in the Real-time CG Content Production" (<http://practical-scheme.net/docs/ILC2002.html>). *Proceedings of the First International Lisp Conference, San Francisco*: 342–348. . Retrieved 2012-08-09
- [47] Bill Magnuson, Hal Abelson, and Mark Friedman (2009-08-11). "Under the Hood of App Inventor for Android" (<http://googleresearch.blogspot.com/2009/08/under-hood-of-app-inventor-for-android.html>). Google Inc, Official Google Research blog. . Retrieved 2012-08-09.

Further reading

- An Introduction to Scheme and its Implementation (ftp://ftp.cs.utexas.edu/pub/garbage/cs345/schintro-v14/schintro_toc.html) (a mirror (http://icem-www.folkwang-hochschule.de/~finnendahl/cm_kurse/doc/schintro/schintro_toc.html))
- Christopher T. Haynes (1999-06-22). "The Scheme Programming Language Standardization Experience" (<http://acm.org/tsc/sstd.html>).
- Guy L. Steele, Jr., Richard P. Gabriel. "The Evolution of Lisp" (<http://www.dreamsongs.org/Files/HOPL2-Uncut.pdf>) (PDF).
- Gerald Sussman and Guy Steele. *SCHEME: An Interpreter for Extended Lambda Calculus* AI Memo 349 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.128.80>), MIT Artificial Intelligence Laboratory, Cambridge, Massachusetts, December 1975.

External links

- The latest version of the Scheme standard: R6RS (<http://www.r6rs.org>)
- A tutorial for new Scheme programmers (<http://www.ccs.neu.edu/home/dorai/t-y-scheme/t-y-scheme.html>), the text of Teach Yourself Scheme in Fixnum Days by Dorai Sitaram
- Scheme (<http://www.dmoz.org/Computers/Programming/Languages/Lisp/Scheme/>) at the Open Directory Project
- Scheme Requests for Implementation (SRFI) (<http://srfi.schemers.org/>)
- Schemers.org (<http://www.schemers.org/>)
- A Tour of Scheme in Gambit (http://dynamo.iro.umontreal.ca/~gambit/wiki/index.php/A_Tour_of_Scheme_in_Gambit), introduction on how to do software development in Gambit Scheme for people with experiences in general programming languages.
- Learning Scheme R6RS Using the DrRacket IDE (<http://ted-gao.blogspot.com/2011/08/learning-scheme-using-drracket.html>)
- Bibliography of Scheme-related research (<http://library.readscheme.org/>)
- Concrete Abstractions : An Introduction to Computer Science Using Scheme (http://www.freebookzone.com/goto.php?bkidx=31&bkcls=pl_scrpt&lkidx=2)

Closer to Math

Type inference

<noinclude Mm q q q Vd XT9 </noinclude> **Type inference** refers to the automatic deduction of the type of an expression in a programming language. If some, but not all, type annotations are already present it is referred to as **type reconstruction**. The opposite operation of type inference is called type erasure.

It is a feature present in some strongly statically typed languages. It is often characteristic of, but not limited to, functional programming languages in general. Some languages that include type inference are ML, OCaml, Haskell, Scala, D, Clean, Opa and Go. It has lately been added (to some extent) to Visual Basic (starting with version 9.0), C# (starting with version 3.0) and C++11. It is also planned for Perl 6. The ability to infer types automatically makes many programming tasks easier, leaving the programmer free to omit type annotations while still permitting type checking.

Nontechnical explanation

In most programming languages, all values have a type explicitly declared at compile time, limiting the values a particular expression can take on at run-time. Increasingly, just-in-time compilation renders the distinction between run time and compile time moot. However, historically, if the type of a value is known only at run-time; these languages are dynamically typed. In other languages, the type of an expression is known only at compile time; these languages are statically typed. In statically typed languages, the input and output types of functions and local variables ordinarily must be explicitly provided by type annotations. For example, in C:

```
int addone(int x) {
    int result; /*declare integer result (C language) */

    result = x + 1;
    return result;
}
```

The signature of this function definition, `int addone(int x)`, declares that `addone` is a function that takes one argument, an integer, and returns an integer. `int result;` declares that the local variable `result` is an integer. In a hypothetical language supporting type inference, the code might be written like this instead:

```
addone(x) {
    val result; /*inferred-type result */
    val result2; /*inferred-type result #2 */

    result = x + 1;
    result2 = x + 1.0; /* this line won't work (in the proposed
language) */
    return result;
}
```

This looks very similar to how code is written in a dynamically typed language, but with some extra constraints (described below) it would be possible to *infer* the types of all the variables at compile time. In the example above,

the compiler would infer that `result` and `x` have type integer and `addone` is a function `int -> int`. The variable `result2` isn't used in a legal manner, so it wouldn't have a type.

In the imaginary language in which the last example is written, the compiler would assume that, in the absence of information to the contrary, `+` takes two integers and returns one integer. (This is how it works in, for example, OCaml). From this, the type inferencer can infer that the type of `x + 1` is an integer, which means `result` is an integer and thus the return value of `addone` is an integer. Similarly, since `+` requires that both of its arguments be of the same type, `x` must be an integer, and therefore `addone` accepts one integer as an argument.

However, in the subsequent line, `result2` is calculated by adding a decimal "1.0" with floating-point arithmetic, causing a conflict in the use of `x` for both integer and floating-point expressions. The correct type-inference algorithm for such a situation has been known since 1958 and has been known to be correct since 1982. It revisits the prior inferences and utilizes the most general type from the outset: in this case floating-point. Frequently, however, degenerate type-inference algorithms are used that are incapable of backtracking and instead generate an error message in such a situation. An algorithm of intermediate generality implicitly declares `result2` as a floating-point variable, and the addition implicitly converts `x` to a floating point. This can be correct if the calling contexts never supply a floating point argument. Such a situation shows the difference between *type inference*, which does not involve type conversion, and *implicit type conversion*, which forces data to a different data type, often without restrictions.

The recent emergence of just-in-time compilation allows for hybrid approaches where the type of arguments supplied by the various calling context is known at compile time, and can generate a large number of compiled versions of the same function. Each compiled version can then be optimized for a different set of types. For instance, JIT compilation allows there to be at least two compiled versions of `addone`:

A version that accepts an integer input and uses implicit type conversion.

A version that accepts a floating-point number as input and utilizes floating point instructions throughout.

Technical description

Type inference is the ability to automatically deduce, either partially or fully, the type of an expression at compile time. The compiler is often able to infer the type of a variable or the type signature of a function, without explicit type annotations having been given. In many cases, it is possible to omit type annotations from a program completely if the type inference system is robust enough, or the program or language is simple enough.

To obtain the information required to infer the type of an expression, the compiler either gathers this information as an aggregate and subsequent reduction of the type annotations given for its subexpressions, or through an implicit understanding of the type of various atomic values (e.g. `true : Bool`; `42 : Integer`; `3.14159 : Real`; etc.). It is through recognition of the eventual reduction of expressions to implicitly typed atomic values that the compiler for a type inferring language is able to compile a program completely without type annotations. In the case of complex forms of higher-order programming and polymorphism, it is not always possible for the compiler to infer as much, however, and type annotations are occasionally necessary for disambiguation.

Example

For example, let us consider the Haskell function `map`, which applies a function to each element of a list, and may be defined as:

```
map f [] = []
map f (first:rest) = f first : map f rest
```

From this, it is evident that the function `map` takes a list as its second argument, that its first argument `f` is a function that can be applied to the type of elements of that list, and that the result of `map` is constructed as a list with elements that are results of `f`. So assuming that a list contains elements of the same type, we can reliably construct a type signature

```
map :: (a -> b) -> [a] -> [b]
```

where the syntax "`a -> b`" denotes a function that takes an `a` as its parameter and produces a `b`. "`a -> b -> c`" is equivalent to "`a -> (b -> c)`".

Note that the inferred type of `map` is parametrically polymorphic: The type of the arguments and results of `f` are not inferred, but left as type variables, and so `map` can be applied to functions and lists of various types, as long as the actual types match in each invocation.

Hindley–Milner type inference algorithm

The algorithm first used to perform type inference is now informally referred to as the Hindley–Milner algorithm, although the algorithm should properly be attributed to Damas and Milner.^[1]

The origin of this algorithm is the type inference algorithm for the simply typed lambda calculus, which was devised by Haskell Curry and Robert Feys in 1958. In 1969 J. Roger Hindley extended this work and proved that their algorithm always inferred the most general type. In 1978 Robin Milner,^[2] independently of Hindley's work, provided an equivalent algorithm, Algorithm W. In 1982 Luis Damas^[1] finally proved that Milner's algorithm is complete and extended it to support systems with polymorphic references.

References

- [1] Damas, Luis; Milner, Robin (1982), "Principal type-schemes for functional programs" (<http://groups.csail.mit.edu/pag/6.883/readings/p207-damas.pdf>), *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ACM, pp. 207–212,
- [2] Milner, Robin (1978), "A Theory of Type Polymorphism in Programming", *Jcss* **17**: 348–375

External links

- Archived e-mail message (<http://www.cis.upenn.edu/~bcpierce/types/archives/1988/msg00042.html>) by Roger Hindley, explains history of type inference
- Polymorphic Type Inference (<http://www.brics.dk/~mis/typeinf.pdf>) by Michael Schwartzbach, gives an overview of Polymorphic type inference.
- Basic Typechecking (<http://lucacardelli.name/Papers/BasicTypechecking.pdf>) paper by Luca Cardelli, describes algorithm, includes implementation in Modula-2
- Implementation (<http://dysphoria.net/2009/06/28/hindley-milner-type-inference-in-scala/>) of Hindley–Milner type inference in Scala, by Andrew Forrest (retrieved July 30, 2009)
- Implementation of Hindley–Milner in Perl 5, by Nikita Borisov (<http://web.archive.org/20070218103011/http://www.cs.berkeley.edu/~nikitab/courses/cs263/hm.html>) at the Wayback Machine (archived February 18, 2007)

- What is Hindley-Milner? (and why is it cool?) (<http://www.codecommit.com/blog/scala/what-is-hindley-milner-and-why-is-it-cool>) Explains Hindley-Milner, examples in Scala

Currying

In mathematics and computer science, **currying** is the technique of transforming a function that takes multiple arguments (or an n -tuple of arguments) in such a way that it can be called as a chain of functions each with a single argument (partial application). It was originated by Moses Schönfinkel^[1] and later re-discovered by Haskell Curry.^{[2][3]} Because of this, some say it would be more correct to name it *schönfinkeling*.^{[4][5]}

Uncurrying is the dual transformation to currying, and can be seen as a form of defunctionalization. It takes a function $f(x)$ which returns another function $g(y)$ as a result, and yields a new function $f(x, y)$ which takes a number of additional parameters and applies them to the function returned by f . The process can be iterated if necessary.

Motivation

Currying is similar to the process of calculating a function of multiple variables for some given values on paper. For example, given the function $f(x,y)=y/x$:

To evaluate $f(2,3)$, first replace x with 2.

Since the result is a function of y , this function $g(y)$ can be defined as $g(y)=f(2,y)=y/2$.

Next, replace the y argument with 3, producing $g(3)=f(2,3)=3/2$.

On paper, using classical notation, this is usually done all in one step. However, each argument can be replaced sequentially as well. Each replacement results in a function taking exactly one argument. This produces a chain of functions as in lambda calculus, and multi-argument functions are usually represented in curried form.

Some programming languages almost always use curried functions to achieve multiple arguments; notable examples are ML and Haskell, where in both cases all functions have exactly one argument.

If we let f be a function

$$f(x, y) = \frac{y}{x}$$

then the function h where

$$h(x) = y \mapsto f(x, y)$$

is a curried version of f . Here, $y \mapsto z$ is a function that maps an argument y to result z . In particular,

$$g(y) = h(2) = y \mapsto f(2, y)$$

is the curried equivalent of the example above. Note, however, that currying, while similar, is not the same operation as partial function application.

Definition

Given a function f of type $f:(X \times Y) \rightarrow Z$, **currying** it makes a function $\text{curry}(f): X \rightarrow (Y \rightarrow Z)$. That is, $\text{curry}(f)$ takes an argument of type X and returns a function of type $Y \rightarrow Z$. **Uncurrying** is the reverse transformation, and is most easily understood in terms of its right adjoint, apply.

The \rightarrow operator is often considered right-associative, so the curried function type $X \rightarrow (Y \rightarrow Z)$ is often written as $X \rightarrow Y \rightarrow Z$. Conversely, function application is considered to be left-associative, so that $f(x, y)$ is equivalent to $\text{curry}(f)(x)y$.

Curried functions may be used in any language that supports closures; however, uncurried functions are generally preferred for efficiency reasons, since the overhead of partial application and closure creation can then be avoided

for most function calls.

Mathematical view

In theoretical computer science, currying provides a way to study functions with multiple arguments in very simple theoretical models such as the lambda calculus in which functions only take a single argument.

In a set-theoretic paradigm, currying is the natural correspondence between the set $A^{B \times C}$ of functions from $B \times C$ to A , and the set $(A^C)^B$ of functions from B to the set of functions from C to A . In category theory, currying can be found in the universal property of an exponential object, which gives rise to the following adjunction in cartesian closed categories: There is a natural isomorphism between the morphisms from a binary product $f:(X \times Y) \rightarrow Z$ and the morphisms to an exponential object $g:X \rightarrow Z^Y$. In other words, currying is the statement that product and Hom are adjoint functors; that is there is a natural transformation:

$$\hom(A \times B, C) \cong \hom(A, C^B).$$

This is the key property of being a Cartesian closed category, and more generally, a closed monoidal category.^[6] The latter, though more rarely discussed, is interesting, as it is the suitable setting for quantum computation, whereas the former is sufficient for classical logic. The difference is that the Cartesian product can be interpreted simply as a pair of items (or a list), whereas the tensor product, used to define a monoidal category, is suitable for describing entangled quantum states.^[7]

Under the Curry–Howard correspondence, the existence of currying and uncurrying is equivalent to the logical theorem $(A \wedge B) \rightarrow C \Leftrightarrow A \rightarrow (B \rightarrow C)$, as tuples (product type) corresponds to conjunction in logic, and function type corresponds to implication.

Curry is a continuous function in the Scott topology.^[8]

Naming

The name "currying", coined by Christopher Strachey in 1967, is a reference to logician Haskell Curry. The alternative name "Schönfinkelisation", has been proposed as a reference to Moses Schönfinkel.^[9]

Contrast with partial function application

Currying and partial function application are often conflated.^[10] The difference between the two is clearest for functions taking more than two arguments.

Given a function of type $f:(X \times Y \times Z) \rightarrow N$, currying produces $\text{curry}(f): X \rightarrow (Y \rightarrow (Z \rightarrow N))$. That is, while an evaluation of the first function might be represented as $f(1,2,3)$, evaluation of the curried function would be represented as $f_{\text{curried}}(1)(2)(3)$, applying each argument in turn to a single-argument function returned by the previous invocation. Note that after calling $f_{\text{curried}}(1)$, we are left with a function that takes a single argument and returns another function, not a function that takes two arguments.

In contrast, **partial function application** refers to the process of fixing a number of arguments to a function, producing another function of smaller arity. Given the definition of f above, we might fix (or 'bind') the first argument, producing a function of type $\text{partial}(f): (Y \times Z) \rightarrow N$. Evaluation of this function might be represented as $f_{\text{partial}}(2,3)$. Note that the result of partial function application in this case is a function that takes two arguments.

Intuitively, partial function application says "if you fix the first arguments of the function, you get a function of the remaining arguments". For example, if function div stands for the division operation x/y , then div with the parameter x fixed at 1 (i.e., $\text{div } 1$) is another function: the same as the function inv that returns the multiplicative inverse of its argument, defined by $\text{inv}(y) = 1/y$.

The practical motivation for partial application is that very often the functions obtained by supplying some but not all of the arguments to a function are useful; for example, many languages have a function or operator similar to

`plus_one`. Partial application makes it easy to define these functions, for example by creating a function that represents the addition operator with 1 bound as its first argument.

Notes

- [1] Strachey, Christopher (2000). "Fundamental Concepts in Programming Languages". *Higher-Order and Symbolic Computation* **13**: 11–49. doi:10.1023/A:101000313106. "There is a device originated by Schönfinkel, for reducing operators with several operands to the successive application of single operand operators." (Reprinted lecture notes from 1967.)
- [2] Henk Barendregt, Erik Barendsen, "Introduction to Lambda Calculus (<ftp://ftp.cs.ru.nl/pub/CompMath.Found/lambd.pdf>)", March 2000, page 8.
- [3] Curry, Haskell; Feys, Robert (1968). *Combinatory logic*. I (2 ed.). Amsterdam, Netherlands: North-Holland.
- [4] Reynolds, John C. (1998). "Definitional Interpreters for Higher-Order Programming Languages". *Higher-Order and Symbolic Computation* **11** (4): 374. doi:10.1023/A:1010027404223. "In the last line we have used a trick called Currying (after the logician H. Curry) to solve the problem of introducing a binary operation into a language where all functions must accept a single argument. (The referee comments that although "Currying" is tastier, "Schönfinkeling" might be more accurate.)"
- [5] Kenneth Slonberger and Barry L. Kurtz. *Formal Syntax and Semantics of Programming Languages*. p. 144.
- [6] Currying (<http://ncatlab.org/nlab/show/currying>) in *nLab*
- [7] John c. Baez and Mike Stay, "Physics, Topology, Logic and Computation: A Rosetta Stone (<http://math.ucr.edu/home/baez/rosetta/rose3.pdf>)", (2009) ArXiv 0903.0340 (<http://arxiv.org/abs/0903.0340/>) in *New Structures for Physics*, ed. Bob Coecke, *Lecture Notes in Physics* vol. **813**, Springer, Berlin, 2011, pp. 95–174.
- [8] Barendregt, H.P. (1984). *The Lambda Calculus*. North-Holland. ISBN 0-444-87508-5. (See theorems 1.2.13, 1.2.14)
- [9] I. Heim and A. Kratzer (1998). *Semantics in Generative Grammar*. Blackwell.
- [10] Partial Function Application is not Currying (http://www.uncarved.com/blog/not_currying.mrk)

References

- Schönfinkel, Moses (1924). "Über die Bausteine der mathematischen Logik". *Math. Ann.* **92** (3–4): 305–316. doi:10.1007/BF01448013.
- Heim, Irene; Kratzer, Angelika (1998). *Semantics in a Generative Grammar*. Malden: Blackwell Publishers

External links

- Frequently Asked Questions for comp.lang.functional: Currying (<http://www.cs.nott.ac.uk/~gmh/faq.html#currying>) by Graham Hutton
- Currying Schonfinkelling (<http://c2.com/cgi/wiki?CurryingSchonfinkelling>) at the Portland Pattern Repository
- Currying != Generalized Partial Application! (<http://lambda-the-ultimate.org/node/2266>) - post at Lambda-the-Ultimate.org

ML (programming language)

ML

Paradigm(s)	multi-paradigm: imperative, functional
Appeared in	1973
Designed by	Robin Milner & others at the University of Edinburgh
Typing discipline	static, strong, inferred
Dialects	Standard ML, OCaml
Influenced by	ISWIM
Influenced	Miranda, Haskell, Cyclone, C++, F#, Clojure, Felix, Opa, Erlang

ML is a general-purpose functional programming language developed by Robin Milner and others in the early 1970s at the University of Edinburgh,^[1] whose syntax is inspired by ISWIM. Historically, ML stands for *metalanguage*: it was conceived to develop proof tactics in the LCF theorem prover (whose language, *pplambda*, a combination of the first-order predicate calculus and the simply typed polymorphic lambda calculus, had ML as its metalanguage). It is known for its use of the Hindley–Milner type inference algorithm, which can automatically infer the types of most expressions without requiring explicit type annotations.

Overview

ML is often referred to as an *impure* functional language, because it encapsulates side-effects, unlike purely functional programming languages such as Haskell.

Features of ML include a call-by-value evaluation strategy, first-class functions, automatic memory management through garbage collection, parametric polymorphism, static typing, type inference, algebraic data types, pattern matching, and exception handling.

Unlike Haskell, ML uses eager evaluation, which means that all subexpressions are always evaluated. However, lazy evaluation can be achieved through the use of closures. Thus one can create and use infinite streams as in Haskell, but their expression is comparatively indirect.

Today there are several languages in the ML family; the two major dialects are Standard ML (SML) and Caml, but others exist, including F# — a language that Microsoft supports for their .NET platform. Ideas from ML have influenced numerous other languages, like Haskell, Cyclone , and Nemerle.

ML's strengths are mostly applied in language design and manipulation (compilers, analyzers, theorem provers), but it is a general-purpose language also used in bioinformatics, financial systems, and applications including a genealogical database, a peer-to-peer client/server program, etc.

ML uses static scoping rules.

Examples

The following examples use the syntax of Standard ML. The other most widely-used ML dialect, OCaml, differs in various insubstantial ways.

Factorial

The factorial function expressed as pure ML:

```
fun fac (0 : int) : int = 1
| fac (n : int) : int = n * fac (n - 1)
```

This describes the factorial as a recursive function, with a single terminating base case. It is similar to the descriptions of factorials found in mathematics textbooks. Much of ML code is similar to mathematics in facility and syntax.

Part of the definition shown is optional, and describes the *types* of this function. The notation $E : t$ can be read as *expression E has type t*. For instance, the argument n is assigned type *integer* (int), and $\text{fac } (n : \text{int})$, the result of applying fac to the integer n , also has type integer. The function fac as a whole then has type *function from integer to integer* (int \rightarrow int). Thanks to type inference, the type annotations can be omitted and will be derived by the compiler. Rewritten without the type annotations, the example looks like:

```
fun fac 0 = 1
| fac n = n * fac (n - 1)
```

The function also relies on pattern matching, an important part of ML programming. Note that parameters of a function are not necessarily in parentheses but separated by spaces. When the function's argument is 0 (zero) it will return the integer 1 (one). For all other cases the second line is tried. This is the recursion, and executes the function again until the base case is reached.

This implementation of the factorial function is not guaranteed to terminate, since a negative argument causes an infinite descending chain of recursive calls. A more robust implementation would check for a nonnegative argument before recursing, as follows:

```
fun fact n = let
  fun fac 0 = 1
  | fac n = n * fac (n - 1)
in
  if (n < 0) then raise Fail "negative argument"
  else fac n
end
```

The problematic case (when n is negative) demonstrates a use of ML's exception system.

The function can be improved further by writing its inner loop in a tail-recursive style, such that the call stack need not grow in proportion to the number of function calls. This is achieved by adding an extra, "accumulator", parameter to the inner function. At last, we arrive at

```
fun factorial n = let
  fun fac (0, acc) = acc
  | fac (n, acc) = fac (n - 1, n*acc)
in
  if (n < 0) then raise Fail "negative argument"
  else fac (n, 1)
end
```

List reverse

The following function "reverses" the elements in a list. More precisely, it returns a new list whose elements are in reverse order compared to the given list.

```
fun reverse [] = []
| reverse (x::xs) = (reverse xs) @ [x]
```

This implementation of reverse, while correct and clear, is inefficient, requiring quadratic time for execution (as ML lists are singly-linked). The function can be rewritten to execute in linear time in the following more efficient, though less easy-to-read, style:

```
fun reverse xs = let
  fun rev [] acc = acc
  | rev (hd::tl) acc = rev tl (hd::acc)
in
  rev xs []
end
```

Notably, this function is an example of parametric polymorphism. That is, it can consume lists whose elements have any type, and return lists of the same type.

Books

- *The Definition of Standard ML*, Robin Milner, Mads Tofte, Robert Harper, MIT Press 1990; (Revised edition adds author David MacQueen), MIT Press 1997. ISBN 0-262-63181-4
- *Commentary on Standard ML*, Robin Milner, Mads Tofte, MIT Press 1997. ISBN 0-262-63137-7
- *ML for the Working Programmer*, L.C. Paulson, Cambridge University Press 1991, 1996, ISBN 0-521-57050-6
- Robert Harper: "Programming in Standard ML", Carnegie Mellon University, 2005. ^[2]
- *Elements of ML Programming*, Jeffrey D. Ullman, Prentice-Hall 1994, 1998. ISBN 0-13-790387-1

References

- [1] Gordon, Michael J. C. (1996). "From LCF to HOL: a short history" (<http://www.cl.cam.ac.uk/~mjcg/papers/HolHistory.html>). .
Retrieved 2007-10-11.
- [2] <http://www.cs.cmu.edu/~rwh/smlbook/book.pdf>

External links

- Standard ML of New Jersey, another popular implementation (<http://smlnj.sf.net/>)
- F#, an ML implementation using the Microsoft .NET framework (<http://msdn.microsoft.com/en-us/fsharp/default.aspx>)
- MLton, a whole-program optimizing Standard ML compiler (<http://mlton.org/>)
- Mythryl, "SML with a Posix face" (<http://mythryl.org/>)
- sML, Successor ML (<http://successor-ml.org/>)

Standard ML

Standard ML

Paradigm(s)	multi-paradigm: functional, imperative
Typing discipline	strong, static, inferred
Major implementations	MLKit, MLton, MLWorks, Moscow ML, Poly/ML, SML/NJ, MLj, SML.NET
Dialects	Alice, Dependent ML
Influenced by	ML, Hope

Standard ML (SML) is a general-purpose, modular, functional programming language with compile-time type checking and type inference. It is popular among compiler writers and programming language researchers, as well as in the development of theorem provers.

SML is a modern descendant of the ML programming language used in the Logic for Computable Functions (LCF) theorem-proving project. It is distinctive among widely used languages in that it has a formal specification, given as typing rules and operational semantics in *The Definition of Standard ML* (1990, revised and simplified as *The Definition of Standard ML (Revised)* in 1997).^[1]

Language

Standard ML is a functional programming language with some impure features. Programs written in Standard ML consist of expressions to be evaluated, as opposed to statements or commands, although some expressions return a trivial "unit" value and are only evaluated for their side-effects.

Like all functional programming languages, a key feature of Standard ML is the function, which is used for abstraction. For instance, the factorial function can be expressed as:

```
fun factorial n =
  if n = 0 then 1 else n * factorial (n-1)
```

A Standard ML compiler is required to infer the static type `int -> int` of this function without user-supplied type annotations. I.e., it has to deduce that `n` is only used with integer expressions, and must therefore itself be an integer, and that all value-producing expressions within the function return integers.

The same function can be expressed with clausal function definitions where the *if-then-else* conditional is replaced by a sequence of templates of the factorial function evaluated for specific values, separated by '|', which are tried one by one in the order written until a match is found:

```
fun factorial 0 = 1
| factorial n = n * factorial (n - 1)
```

This can be rewritten using a case statement like this:

```
val rec factorial =
  fn n => case n of 0 => 1
  | n => n * factorial (n - 1)
```

or as a lambda function:

```
val rec factorial = fn 0 => 1 | n => n * factorial(n - 1)
```

Here, the keyword `val` introduces a binding of an identifier to a value, `fn` introduces the definition of an anonymous function, and `case` introduces a sequence of patterns and corresponding expressions.

Using a local function, this function can be rewritten in a more efficient tail recursive style.

```
fun factorial n = let
  fun lp (0, acc) = acc
  | lp (m, acc) = lp (m-1, m*acc)
in
  lp (n, 1)
end
```

(The value of a `let`-expression is that of the expression between `in` and `end`.) The encapsulation of an invariant-preserving tail-recursive tight loop with one or more accumulator parameters inside an invariant-free outer function, as seen here, is a common idiom in Standard ML, and appears with great frequency in SML code.

Type synonyms

A type synonym is defined with the `type` keyword. Here is a type synonym for points in the plane, and functions computing the distances between two points, and the area of a triangle with the given corners as per Heron's formula.

```
type loc = real * real

fun dist ((x0, y0), (x1, y1)) = let
  val dx = x1 - x0
  val dy = y1 - y0
  in
    Math.sqrt (dx * dx + dy * dy)
  end

fun heron (a, b, c) = let
  val ab = dist (a, b)
  val bc = dist (b, c)
  val ac = dist (a, c)
  val perim = ab + bc + ac
  val s = perim / 2.0
  in
    Math.sqrt (s * (s - ab) * (s - bc) * (s - ac))
  end
```

Algebraic datatypes and pattern matching

Standard ML provides strong support for algebraic datatypes. An ML datatype can be thought of as a disjoint union. They are easy to define and easy to program with, in large part because of Standard ML's pattern matching as well as most Standard ML implementations' pattern exhaustiveness checking and pattern redundancy checking.

A datatype is defined with the `datatype` keyword, as in

```
datatype shape
  = Circle  of loc * real      (* center and radius *)
  | Square   of loc * real      (* upper-left corner and side length; axis-aligned *)
  | Triangle of loc * loc * loc (* corners *)
```

(See above for the definition of `loc`.) Note: datatypes, not type synonyms, are necessary to define recursive constructors. (This is not at issue in the present example.)

Order matters in pattern matching; patterns that are textually first are tried first. Pattern matching can be syntactically embedded in function definitions as follows:

```
fun area (Circle (_, r)) = 3.14 * r * r
  | area (Square (_, s)) = s * s
  | area (Triangle (a, b, c)) = heron (a, b, c) (* see above *)
```

Note that subcomponents whose values are not needed in a particular computation are ellided with underscores, or so-called wildcard patterns.

The so-called "clausal form" style function definition, where patterns appear immediately after the function name, is merely syntactic sugar for

```
fun area shape =
  case shape
    of Circle (_, r) => 3.14 * r * r
    | Square (_, s) => s * s
    | Triangle (a, b, c) => heron (a, b, c)
```

Pattern exhaustiveness checking will make sure each case of the datatype has been accounted for, and will produce a warning if not. The following pattern is inexhaustive:

```
fun center (Circle (c, _)) = c
  | center (Square ((x, y), s)) = (x + s / 2.0, y + s / 2.0)
```

There is no pattern for the `Triangle` case in the `center` function. The compiler will issue a warning that the pattern is inexhaustive, and if, at runtime, a `Triangle` is passed to this function, the exception `Match` will be raised.

The set of clauses in the following function definition is exhaustive and not redundant:

```
fun hasCorners (Circle _) = false
  | hasCorners _ = true
```

If control gets past the first pattern (the `Circle`), we know the value must be either a `Square` or a `Triangle`. In either of those cases, we know the shape has corners, so we can return `true` without discriminating which case we are in.

The pattern in second clause the following (meaningless) function is redundant:

```
fun f (Circle ((x, y), r)) = x+y
  | f (Circle _) = 1.0
  | f _ = 0.0
```

Any value that matches the pattern in the second clause will also match the pattern in the first clause, so the second clause is unreachable. Therefore this definition as a whole exhibits redundancy, and causes a compile-time warning.

C programmers will often use tagged unions, dispatching on tag values, to accomplish what ML accomplishes with datatypes and pattern matching. Nevertheless, while a C program decorated with appropriate checks will be in a sense as robust as the corresponding ML program, those checks will of necessity be dynamic; ML provides a set of static checks that give the programmer a high degree of confidence in the correctness of the program at compile time.

Note that in object-oriented programming languages, such as Java, a disjoint union can be expressed by designing class hierarchies.

Higher-order functions

Functions can consume functions as arguments:

```
fun applyToBoth f x y = (f x, f y)
```

Functions can produce functions as return values:

```
fun constantFn k = let
  fun const anything = k
  in
    const
  end
```

(alternatively)

```
fun constantFn k = (fn anything => k)
```

Functions can also both consume and produce functions:

```
fun compose (f, g) = let
  fun h x = f (g x)
  in
    h
  end
```

(alternatively)

```
fun compose (f, g) = (fn x => f (g x))
```

The function `List.map` from the basis library is one of the most commonly used higher-order functions in Standard ML:

```
fun map _ [] = []
| map f (x::xs) = f x :: map f xs
```

(A more efficient implementation of `map` would define a tail-recursive inner loop as follows:)

```
fun map f xs = let
  fun m ([] , acc) = List.rev acc
  | m (x::xs, acc) = m (xs, f x :: acc)
  in
    m (xs, [])
  end
```

Exceptions

Exceptions are raised with the `raise` keyword, and handled with pattern matching `handle` constructs.

```
exception Undefined
fun max [x] = x
| max (x::xs) = let val m = max xs in if x > m then x else m end
| max [] = raise Undefined
fun main xs = let
  val msg = (Int.toString (max xs)) handle Undefined => "empty list...there is no max!"
  in
  print (msg ^ "\n")
end
```

The exception system can be exploited to implement non-local exit, an optimization technique suitable for functions like the following.

```
exception Zero
fun listProd ns = let
  fun p [] = 1
  | p (0::_) = raise Zero
  | p (h::t) = h * p t
  in
  (p ns) handle Zero => 0
end
```

When the exception `Zero` is raised in the 0 case, control leaves the function `p` altogether. Consider the alternative: the value 0 would be returned to the most recent awaiting frame, it would be multiplied by the local value of `h`, the resulting value (inevitably 0) would be returned in turn to the next awaiting frame, and so on. The raising of the exception allows control to leapfrog directly over the entire chain of frames and avoid the associated computation.

Module system

Standard ML has an advanced module system, allowing programs to be decomposed into hierarchically organized *structures* of logically related type and value declarations. SML modules provide not only namespace control but also abstraction, in the sense that they allow programmers to define abstract data types.

Three main syntactic constructs comprise the SML module system: signatures, structures and functors. A *structure* is a module; it consists of a collection of types, exceptions, values and structures (called *substructures*) packaged together into a logical unit. A *signature* is an interface, usually thought of as a type for a structure: it specifies the names of all the entities provided by the structure as well as the arities of type components, the types of value components, and signatures for substructures. The definitions of type components may or may not be exported; type components whose definitions are hidden are *abstract types*. Finally, a *functor* is a function from structures to structures; that is, a functor accepts one or more arguments, which are usually structures of a given signature, and produces a structure as its result. Functors are used to implement generic data structures and algorithms.

For example, the signature for a queue data structure might be:

```
signature QUEUE =
sig
  type 'a queue
  exception Queue
  val empty : 'a queue
```

```

val isEmpty    : 'a queue -> bool
val singleton : 'a -> 'a queue
val insert     : 'a * 'a queue -> 'a queue
val peek       : 'a queue -> 'a
val remove    : 'a queue -> 'a * 'a queue
end

```

This signature describes a module that provides a parameterized type `queue` of queues, an exception called `Queue`, and six values (five of which are functions) providing the basic operations on queues. One can now implement the queue data structure by writing a structure with this signature:

```

structure TwoListQueue :> QUEUE =
struct

  type 'a queue = 'a list * 'a list
  exception Queue

  val empty = ([], [])

  fun isEmpty ([] , []) = true
    | isEmpty _ = false

  fun singleton a = ([a], [])

  fun insert (a, (ins,outs)) = (a::ins,outs)

  fun peek ([] , []) = raise Queue
    | peek (ins, []) = hd (rev ins)
    | peek (ins, a::outs) = a

  fun remove ([] , []) = raise Queue
    | remove (ins, []) =
      let val newouts = rev ins
      in (hd newouts, [], tl newouts)
      end
    | remove (ins, a::outs) = (a, (ins,outs))

end

```

This definition declares that `TwoListQueue` is an implementation of the `QUEUE` signature. Furthermore, the *opaque ascription* (denoted by `:>`) states that any type components whose definitions are not provided in the signature (*i.e.*, `queue`) should be treated as abstract, meaning that the definition of a queue as a pair of lists is not visible outside the module. The body of the structure provides bindings for all of the components listed in the signature.

To use a structure, one can access its type and value members using "dot notation". For instance, a queue of strings would have type `string TwoListQueue.queue`, the empty queue is `TwoListQueue.empty`, and to remove the first element from a queue called `q` one would write `TwoListQueue.remove q`.

One popular algorithm^[2] for breadth-first traversal of trees makes uses of queues. Here we present a version of that algorithm parameterized over an abstract queue structure:

```

functor BFT (Q: QUEUE) = (* after Okasaki, ICFP, 2000 *)
struct

  datatype 'a tree
  = E
  | T of 'a * 'a tree * 'a tree
  fun bftQ (q : 'a tree Q.queue) : 'a list =
    if Q.isEmpty q then []
    else let
      val (t, q') = Q.remove q
      in case t
        of E => bftQ q'
        | T (x, l, r) => let
          val q'' = Q.insert (r, Q.insert (l, q'))
          in
            x :: bftQ q''
          end
        end
    end
  fun bft t = bftQ (Q.singleton t)
end

```

Please note that inside the `BFT` structure, the program has no access to the particular queue representation in play. More concretely, there is no way for the program to, say, select the first list in the two-list queue representation, if that is indeed the representation being used. This data abstraction mechanism makes the breadth-first code truly agnostic to the queue representation choice. This is in general desirable; in the present case, the queue structure can safely maintain any of the various logical invariants on which its correctness depends behind the bulletproof wall of abstraction.

Code examples

Snippets of SML code are most easily studied by entering them into a "top-level", also known as a read-eval-print loop. This is an interactive session that prints the inferred types of resulting or defined expressions. Many SML implementations provide an interactive top-level, including SML/NJ:

```

$ sml
Standard ML of New Jersey v110.52 [built: Fri Jan 21 16:42:10 2005]
-

```

Code can then be entered at the "-" prompt. For example, to calculate $1+2*3$:

```

- 1 + 2 * 3;
val it = 7 : int

```

The top-level infers the type of the expression to be "int" and gives the result "7".

Hello world

The following program "hello.sml":

```
print "Hello world!\n";
```

can be compiled with MLton:

```
$ mlton hello.sml
```

and executed:

```
$ ./hello
Hello world!
```

Insertion sort

Insertion sort for lists of integers (ascending) is expressed concisely as follows:

```
fun ins (n, []) = [n]
| ins (n, ns as h::t) = if (n < h) then n::ns else h::(ins (n, t))
val insertionSort = List.foldr ins []
```

This can be made polymorphic by abstracting over the ordering operator. Here we use the symbolic name `<<` for that operator.

```
fun ins' << (num, nums) = let
  fun i (n, []) = [n]
  | i (n, ns as h::t) = if <<(n,h) then n::ns else h::i(n,t)
  in
    i (num, nums)
  end
fun insertionSort' << = List.foldr (ins' <<) []
```

The type of `insertionSort'` is `('a * 'a -> bool) -> ('a list) -> ('a list)`.

Mergesort

Here, the classic mergesort algorithm is implemented in three functions: split, merge and mergesort.

The function `split` is implemented with a local function named `loop`, which has two additional parameters. The local function `loop` is written in a tail-recursive style; as such it can be compiled efficiently. This function makes use of SML's pattern matching syntax to differentiate between non-empty list (`x::xs`) and empty list (`[]`) cases. For stability, the input list `ns` is reversed before being passed to `loop`.

```
(* Split list into two near-halves, returned as a pair.
 * The "halves" will either be the same size,
 * or the first will have one more element than the second.
 * Runs in O(n) time, where n = |xs|. *)
local
  fun loop (x::y::zs, xs, ys) = loop (zs, x::xs, y::ys)
  | loop (x::[], xs, ys) = (x::xs, ys)
  | loop ([], xs, ys) = (xs, ys)
in
  fun split ns = loop (List.rev ns, [], [])
```

```
end
```

The local-in-end syntax could be replaced with a let-in-end syntax, yielding the equivalent definition:

```
fun split ns = let
  fun loop (x::y::zs, xs, ys) = loop (zs, x::xs, y::ys)
  | loop ([]:[], xs, ys) = (x::xs, ys)
  | loop ([], xs, ys) = (xs, ys)
in
  loop (List.rev ns, [], [])
end
```

As with split, merge also uses a local function loop for efficiency. The inner `loop` is defined in terms of cases: when two non-empty lists are passed, when one non-empty list is passed, and when two empty lists are passed. Note the use of the underscore (`_`) as a wildcard pattern.

This function merges two "ascending" lists into one ascending list. Note how the accumulator `out` is built "backwards", then reversed with `List.rev` before being returned. This is a common technique—build a list backwards, then reverse it before returning it. In SML, lists are represented as imbalanced binary trees, and thus it is efficient to prepend an element to a list, but inefficient to append an element to a list. The extra pass over the list is a linear time operation, so while this technique requires more wall clock time, the asymptotics are not any worse.

```
(* Merge two ordered lists using the order lt.
 * Pre: the given lists xs and ys must already be ordered per lt.
 * Runs in O(n) time, where n = |xs| + |ys|. *)
fun merge lt (xs, ys) = let
  fun loop (out, left as x::xs, right as y::ys) =
    if lt (x, y) then loop (x::out, xs, right)
    else loop (y::out, left, ys)
  | loop (out, x::xs, []) = loop (x::out, xs, [])
  | loop (out, [], y::ys) = loop (y::out, [], ys)
  | loop (out, [], []) = List.rev out
in
  loop ([]:[], xs, ys)
end
```

The main function.

```
(* Sort a list in according to the given ordering operation lt.
 * Runs in O(n log n) time, where n = |xs|.
 *)
fun mergesort lt xs = let
  val merge' = merge lt
  fun ms [] = []
  | ms [x] = [x]
  | ms xs = let
    val (left, right) = split xs
    in
      merge' (ms left, ms right)
    end
  in
```

```
ms xs
end
```

Also note that the code makes no mention of variable types, with the exception of the :: and [] syntax which signify lists. This code will sort lists of any type, so long as a consistent ordering function lt can be defined. Using Hindley–Milner type inference, the compiler is capable of inferring the types of all variables, even complicated types such as that of the lt function.

Quicksort

Quicksort can be expressed as follows. This generic quicksort consumes an order operator <<.

```
val filt = List.filter
fun quicksort << xs = let
  fun qs [] = []
  | qs [x] = [x]
  | qs (p::xs) = let
    val lessThanP = (fn x => << (x, p))
    in
      qs (filt lessThanP xs) @ p :: (qs (filt (not o lessThanP) xs))
  end
  in
  qs xs
end
```

Expression language

Note the relative ease with which a small expression language is defined and processed.

```
exception Err

datatype ty
= IntTy
| BoolTy

datatype exp
= True
| False
| Int of int
| Not of exp
| Add of exp * exp
| If of exp * exp * exp

fun typeOf (True) = BoolTy
| typeOf (False) = BoolTy
| typeOf (Int _) = IntTy
| typeOf (Not e) = if typeOf e = BoolTy then BoolTy else raise Err
| typeOf (Add (e1, e2)) =
  if (typeOf e1 = IntTy) andalso (typeOf e2 = IntTy) then IntTy else raise Err
| typeOf (If (e1, e2, e3)) =
```

```

if typeOf e1 <> BoolTy then raise Err
else if typeOf e2 <> typeOf e3 then raise Err
else typeOf e2

fun eval (True) = True
| eval (False) = False
| eval (Int n) = Int n
| eval (Not e) =
  (case eval e
   of True => False
   | False => True
   | _ => raise Fail "type-checking is broken")
| eval (Add (e1, e2)) = let
  val (Int n1) = eval e1
  val (Int n2) = eval e2
  in
  Int (n1 + n2)
  end
| eval (If (e1, e2, e3)) =
  if eval e1 = True then eval e2 else eval e3

fun chkEval e = (ignore (typeOf e); eval e) (* will raise Err on type error *)

```

Arbitrary-precision factorial function (libraries)

In SML, the IntInf module provides arbitrary-precision integer arithmetic. Moreover, integer literals may be used as arbitrary-precision integers without the programmer having to do anything.

The following program "fact.sml" implements an arbitrary-precision factorial function and prints the factorial of 120:

```

fun fact n : IntInf.int =
  if n=0 then 1 else n * fact(n - 1)

val () =
  print (IntInf.toString (fact 120) ^ "\n")

```

and can be compiled and run with:

```

$ mlton fact.sml
$ ./fact
66895029134491270575881180540903725867527463331380298102956713523016335
57244962989366874165271984981308157637893214090552534408589408121859898
481114389650005964960521256960000000000000000000000000000000000000000000000000

```

Numerical derivative (higher-order functions)

Since SML is a functional programming language, it is easy to create and pass around functions in SML programs. This capability has an enormous number of applications. Calculating the numerical derivative of a function is one such application. The following SML function "d" computes the numerical derivative of a given function "f" at a given point "x":

```
- fun d delta f x =
  (f (x + delta) - f (x - delta)) / (2.0 * delta);
val d = fn : real -> (real -> real) -> real -> real
```

This function requires a small value "delta". A good choice for delta when using this algorithm is the cube root of the machine epsilon.

The type of the function "d" indicates that it maps a "float" onto another function with the type "(real -> real) -> real -> real". This allows us to partially apply arguments. This functional style is known as currying. In this case, it is useful to partially apply the first argument "delta" to "d", to obtain a more specialised function:

```
- val d = d 1E~8;
val d = fn : (real -> real) -> real -> real
```

Note that the inferred type indicates that the replacement "d" is expecting a function with the type "real -> real" as its first argument. We can compute a numerical approximation to the derivative of $f(x) = x^3 - x - 1$ at $x = 3$ with:

```
- d (fn x => x * x * x - x - 1.0) 3.0;
val it = 25.9999996644 : real
```

The correct answer is $f'(x) = 3x^2 - 1 \Rightarrow f'(3) = 27 - 1 = 26$.

The function "d" is called a "higher-order function" because it accepts another function ("f") as an argument.

Curried and higher-order functions can be used to eliminate redundant code. For example, a library may require functions of type $a \rightarrow b$, but it is more convenient to write functions of type $a * c \rightarrow b$ where there is a fixed relationship between the objects of type a and c . A higher order function of type $(a * c \rightarrow b) \rightarrow (a \rightarrow b)$ can factor out this commonality. This is an example of the adapter pattern.

Discrete wavelet transform (pattern matching)

The 1D Haar wavelet transform of an integer-power-of-two-length list of numbers can be implemented very succinctly in SML and is an excellent example of the use of pattern matching over lists, taking pairs of elements ("h1" and "h2") off the front and storing their sums and differences on the lists "s" and "d", respectively:

```
- fun haar l = let
  fun aux [s] [] d = s :: d
    | aux [] s d = aux s [] d
    | aux (h1::h2::t) s d = aux t (h1+h2 :: s) (h1-h2 :: d)
    | aux _ _ _ = raise Empty
  in
    aux l [] []
  end;
val haar = fn : int list -> int list
```

For example:

```
- haar [1, 2, 3, 4, ~4, ~3, ~2, ~1];
val it = [0,20,4,4,~1,~1,~1,~1] : int list
```

Pattern matching is a useful construct that allows complicated transformations to be represented clearly and succinctly. Moreover, SML compilers turn pattern matches into efficient code, resulting in programs that are not only shorter but also faster.

Implementations

Many SML implementations exist, including:

- MLton is a whole-program optimizing compiler that produces very fast code compared to other ML implementations. [3]
- Poly/ML^[4] is a full implementation of Standard ML that produces fast code and supports multicore hardware (via Posix threads); its runtime system performs parallel garbage collection and online sharing of immutable substructures.
- Isabelle/ML^[5] integrates parallel Poly/ML into an interactive theorem prover, with a sophisticated IDE (based on jEdit) both for ML and the proof language.
- Standard ML of New Jersey (abbreviated SML/NJ) is a full compiler, with associated libraries, tools, an interactive shell, and documentation. [6]
- Moscow ML is a light-weight implementation, based on the CAML Light runtime engine. It implements the full SML language, including SML Modules, and much of the SML Basis Library. [7]
- TILT^[8] is a full certifying compiler for SML. It uses typed intermediate languages to optimize code and ensure correctness, and can compile to typed Assembly language.
- HaMLet^[9] is an SML interpreter that aims to be an accurate and accessible reference implementation of the standard.
- The ML Kit^[10] integrates a garbage collector (which can be disabled) and region-based memory management with automatic inference of regions, aiming realtime applications. Its implementation is based very closely on the Definition.
- SML.NET^[11] allows compiling to the Microsoft CLR and has extensions for linking with other .NET code.
- SML2c is a batch compiler and compiles only module-level declarations (i.e. signatures, structures, functors) into C. It is based on SML/NJ version 0.67 and shares the front end, and most of its run-time system, but does not support SML/NJ style debugging and profiling. Module-level programs that run on SML/NJ can be compiled by sml2c with no changes.
- The Poplog system implements a version of SML, with POP-11, and optionally Common Lisp, and Prolog, allowing mixed language programming. For all, the implementation language is POP-11, which is compiled incrementally. It also has an integrated Emacs-like editor that communicates with the compiler.
- SML#^[12] is an extension of SML providing record polymorphism and C language interoperability. It is a conventional native compiler and its name is *not* an allusion to running on the .NET framework.
- Alice: an interpreter for Standard ML by Saarland University adding features for lazy evaluation, concurrency (multithreading and distributed computing via remote procedure calls) and constraint programming.

All of these implementations are open-source and freely available. Most are implemented themselves in SML. There are no longer any commercial SML implementations. Harlequin once produced a commercial IDE and compiler for SML called MLWorks. The company is now defunct. MLWorks is believed to have been passed on to Xanalys.

References

- [1] Milner, R.; M. Tofte, R. Harper and D. MacQueen. (1997). *The Definition of Standard ML (Revised)*. MIT Press. ISBN 0-262-63181-4.
- [2] Okasaki, Chris (2000). "Breadth-First Numbering: Lessons from a Small Exercise in Algorithm Design". *International Conference on Functional Programming 2000*. ACM.
- [3] <http://www.mltton.org>
- [4] <http://www.polyml.org/>
- [5] <http://isabelle.in.tum.de>
- [6] <http://www.smlnj.org/>
- [7] <http://www.itu.dk/people/sestoft/mosml.html>
- [8] <http://www.tilt.cs.cmu.edu/>
- [9] <http://www.mpi-sws.org/~rossberg/hamlet/>
- [10] <http://www.it-c.dk/research/mlkit/>
- [11] <http://www.cl.cam.ac.uk/Research/TSG/SMLNET/>
- [12] <http://www.pllab.riec.tohoku.ac.jp/smlsharp/>

External links

- What is SML? (<http://www.smlnj.org/sml.html>)
- What is SML '97? (<http://www.smlnj.org/sml97.html>)
- successor ML (sML) (<http://www.successor-ml.org>) is intended to provide a vehicle for the continued evolution of ML, using Standard ML as a starting point.
- Standard ML language (http://www.scholarpedia.org/article/Standard_ML_language) at Scholarpedia, curated by Mads Tofte.

Tail call

In computer science, a **tail call** is a subroutine call that happens inside another procedure as its final action; it may produce a return value which is then immediately returned by the calling procedure. The call site is then said to be in **tail position**, i.e. at the end of the calling procedure. If any call that a subroutine performs, such that it might eventually lead to this same subroutine being called again down the call chain, is in tail position, such a subroutine is said to be **tail-recursive**. This is a special case of recursion.

Tail calls are significant because they can be implemented without adding a new stack frame to the call stack. Most of the frame of the current procedure is not needed any more, and it can be replaced by the frame of the tail call, modified as appropriate (similar to overlay for processes, but for function calls). The program can then jump to the called subroutine. Producing such code instead of a standard call sequence is called **tail call elimination**, or **tail call optimization**.

Traditionally, tail call elimination is optional. However, in functional programming languages, tail call elimination is often guaranteed by the language standard, and this guarantee allows using recursion, in particular tail recursion, in place of loops. In such cases, it is not correct (though it may be customary) to refer to it as an optimization.

Description

When a function is called, the computer must "remember" the place it was called from, the *return address*, so that it can return to that location with the result once the call is complete. Typically, this information is saved on the call stack, a simple list of return locations in order of the times that the call locations they describe were reached. For tail calls, there is no need to remember the place we are calling from — instead, we can perform tail call elimination by leaving the stack alone (except possibly for function arguments and local variables^[1]), and the newly called function will return its result directly to the *original* caller. Note that the tail call doesn't have to appear lexically after all other statements in the source code; it is only important that the calling function return immediately after the tail call,

returning the tail call's result if any, since the calling function will never get a chance to do anything after the call if the optimization is performed.

For non-recursive function calls, this is usually an optimization that saves little time and space, since there are not that many different functions available to call. When dealing with recursive or mutually recursive functions where recursion happens through tail calls, however, the stack space and the number of returns saved can grow to be very significant, since a function can call itself, directly or indirectly, many times. In fact, it often asymptotically reduces stack space requirements from linear, or $O(n)$, to constant, or $O(1)$. Tail call elimination is thus required by the standard definitions of some programming languages, such as Scheme,^{[2][3]} and languages in the ML family among others. In the case of Scheme, the language definition formalizes the intuitive notion of tail position exactly, by specifying which syntactic forms allow having results in tail context.^[4] Implementations allowing an unlimited number of tail calls to be active at the same moment, thanks to tail call elimination, can also be called 'properly tail-recursive'.^[2]

Besides space and execution efficiency, tail call elimination is important in the functional programming idiom known as continuation passing style (CPS), which would otherwise quickly run out of stack space.

Syntactic form

A tail call can be located just before the syntactical end of a subroutine:

```
function foo(data) {
    a(data);
    return b(data);
}
```

Here, both `a(data)` and `b(data)` are calls, but `b` is the last thing the procedure executes before returning and is thus in tail position. However, not all tail calls are necessarily located at the syntactical end of a subroutine. Consider:

```
function bar(data) {
    if ( a(data) ) {
        return b(data);
    }
    return c(data);
}
```

Here, both calls to `b` and `c` are in tail position, even though the first one is not syntactically at the end of `bar`'s body.

Now consider this code:

```
function fool(data) {
    return a(data) + 1;
}

function foo2(data) {
    var ret = a(data);
    return ret;
}

function foo3(data) {
    var ret = a(data);
    return (ret === 0) ? 1 : ret;
```

```
}
```

Here, the call to `a(data)` is in tail position in `foo2`, but it is **not** in tail position either in `foo1` or in `foo3`, because control must return to the caller to allow it to inspect or modify the return value before returning it.

Example programs

Take this Scheme program as an example:

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
(define (factorial n)
  (if (= n 0)
      1
      (* n (factorial (- n 1)))))
```

This program is not written in a tail recursion style. Now take this Scheme program as an example:

```
;; factorial : number -> number
;; to calculate the product of all positive
;; integers less than or equal to n.
(define (factorial n)
  (let fact ([i n] [acc 1])
    (if (zero? i)
        acc
        (fact (- i 1) (* acc i)))))
```

The inner procedure `fact` calls itself *last* in the control flow. This allows an interpreter or compiler to reorganize the execution which would ordinarily look like this:

```
call factorial (3)
call fact (3 1)
call fact (2 3)
call fact (1 6)
call fact (0 6)
return 6
return 6
return 6
return 6
return 6
```

into the more efficient variant, in terms of both space and time:

```
call factorial (3)
call fact (3 1)
replace arguments with (2 3), jump to "fact"
replace arguments with (1 6), jump to "fact"
replace arguments with (0 6), jump to "fact"
return 6
return 6
```

This reorganization saves space because no state except for the calling function's address needs to be saved, either on the stack or on the heap, and the call stack frame for `fact` is reused for the intermediate results storage. This also means that the programmer need not worry about running out of stack or heap space for extremely deep recursions. It is also worth noting, in typical implementations, the tail recursive variant will be substantially faster than the other variant, but only by a constant factor.

Some programmers working in functional languages will rewrite recursive code to be tail-recursive so they can take advantage of this feature. This often requires addition of an "accumulator" argument (`acc` in the above example) to the function. In some cases (such as filtering lists) and in some languages, full tail recursion may require a function that was previously purely functional to be written such that it mutates references stored in other variables.

An example in pseudo-C follows. Suppose we have the following functions:

```
int a(int x, int y)
{
    foobar(x, y);
    return b(x + 1, y + 2);
}

int b(int u, int v)
{
    foobar(u, v);
    return u + v;
}
```

Function `a` can be changed to:

```
int a(int x, int y)
{
    foobar(x, y);
    b:u = a:x + 1;
    b:v = a:y + 2;
    jump b;
}
```

There are possible aliasing problems but this is the basic idea.

Tail recursion modulo cons

Tail recursion modulo cons is a generalization of tail recursion optimization introduced by David H. D. Warren^[5] in the context of compilation of Prolog, seen as an explicitly set-once language. As the name suggests, it applies when the only operation left to perform after a recursive call is to prepend a known value in front of a list returned from it (or to perform a constant number of simple data-constructing operations in general), which would thus be *tail call* save for the said *cons* operation. But prefixing a value at the start of a list *on exit* from a recursive call is the same as appending this value at the end of the growing list *on entry* into the recursive call, thus building the list as a side effect, as if in an implicit accumulator parameter. The following Prolog fragment illustrates the concept:

```
partition([], _, [], []).
% -- Haskell

translation:

partition([X|Xs], Pivot, [X|Rest], Bigs) :-
    = ([], [])
% partition [] _
```

```

X @< Pivot, !,
partition(Xs, Pivot, Rest, Bigs).                                % partition (x:xs) p / x < p = (x:a,b)
| True   = (a,x:b)                                              %
partition([X|Xs], Pivot, Smalls, [X|Rest]) :-                  % where
    partition(Xs, Pivot, Smalls, Rest).                            % (a,b) =
partition xs p

% to be compiled not as this:                                     % but as this:
partition([], _, [], []).                                         partition([], _, [],
[], []).

partition([X|Xs], Pivot, Smalls, Bigs) :-                         (
partition([X|Xs], Pivot, Smalls, Bigs) :-                         X @< Pivot
    partition(Xs, Pivot, Rest, Bigs), Smalls=[X|Rest]           ->
Smalls=[X|Rest], partition(Xs, Pivot, Rest, Bigs)
    ; partition(Xs, Pivot, Smalls, Rest), Bigs=[X|Rest]          ;
Bigs=[X|Rest], partition(Xs, Pivot, Smalls, Rest)
).

```

Thus such a call is transformed into creating a new list node, setting its `first` field, and then making a tail call which is also passed a pointer to where its result should be written (here, the node's `rest` field).

As another example, consider a function in C language that duplicates a linked list:

```

list *duplicate(const list *input)
{
    list *head;
    if (input != NULL) {
        head      = malloc(sizeof *head);
        head->value = input->value;
        head->next = duplicate(input->next);
    } else {
        head = NULL;
    }
    return head;
}

```

In this form the function is not tail-recursive, because control returns to the caller after the recursive call duplicates the rest of input list. Even though it actually allocates the head node prior to duplicating the rest, the caller still has to plug in the result from the callee into the `next` field. So the function is *almost* tail-recursive. Warren's method gives the following purely tail-recursive implementation which passes the `head` node to the callee to have its `next` field set by it:

```

list *duplicate(const list *input)
{
    list head;
    duplicate_aux(input, &head);
    return head.next;
}

```

```

void duplicate_aux(const list *input, list *end)
{
    if (input != NULL) {
        end->next      = malloc(sizeof *end);
        end->next->value = input->value;
        duplicate_aux(input->next, end->next);
    } else {
        end->next      = NULL;
    }
}

```

Note how the callee now appends to the end of the list, rather than have the caller prepend to the beginning. Characteristically for this technique, a parent frame is created here in the execution call stack, which calls (non-tail-recursively) into the tail-recursive callee which could reuse its call frame if the tail-call optimization were present in C, thus defining an iterative computation.

This properly tail-recursive implementation can be converted into explicitly iterative form:

```

list *duplicate(const list *input)
{
    list head, *end;
    for (end = &head; input != NULL; input = input->next, end =
end->next )
    {
        end->next      = malloc(sizeof *end);
        end->next->value = input->value;
    }
    end->next = NULL;
    return head.next;
}

```

History

In a paper delivered to the ACM conference in Seattle in 1977, Guy L. Steele summarized the debate over the GOTO and structured programming, and observed that procedure calls in the tail position of a procedure can be best treated as a direct transfer of control to the called procedure, typically eliminating unnecessary stack manipulation operations.^[6] Since such "tail calls" are very common in Lisp, a language where procedure calls are ubiquitous, this form of optimization considerably reduces the cost of a procedure call compared to other implementations. Steele argued that poorly implemented procedure calls had led to an artificial perception that the GOTO was cheap compared to the procedure call. Steele further argued that "in general procedure calls may be usefully thought of as GOTO statements which also pass parameters, and can be uniformly coded as [machine code] JUMP instructions", with the machine code stack manipulation instructions "considered an optimization (rather than vice versa!)".^[6] Steele cited evidence that well optimized numerical algorithms in Lisp could execute faster than code produced by then-available commercial Fortran compilers because the cost of a procedure call in Lisp was much lower. In Scheme, a Lisp dialect developed by Steele with Gerald Jay Sussman, tail call elimination is mandatory.^[7]

Implementation methods

Tail recursion is important to some high-level languages, especially functional and logic languages and members of the Lisp family. In these languages, tail recursion is the most commonly used way (and sometimes the only way available) of implementing iteration. The language specification of Scheme requires that tail calls are to be optimized so as not to grow the stack. Tail calls can be made explicitly in Perl, with a variant of the "goto" statement that takes a function name: `goto &NAME;`^[8]

Various implementation methods are available.

In assembler

For compilers generating assembly directly, tail call elimination is easy: it suffices to replace a call opcode with a jump one, after fixing parameters on the stack. From a compiler's perspective, the first example above is initially translated into pseudo-assembly language:

```
foo:  
    call B  
    call A  
    ret
```

Tail call elimination replaces the last two lines with a single jump instruction:

```
foo:  
    call B  
    jmp A
```

After subroutine A completes, it will then return directly to the return address of foo, omitting the unnecessary `ret` statement.

Typically, the subroutines being called need to be supplied with parameters. The generated code thus needs to make sure that the call frame for A is properly set up before jumping to the tail-called subroutine. For instance, on platforms where the call stack does not just contain the return address, but also the parameters for the subroutine, the compiler may need to emit instructions to adjust the call stack. On such a platform, consider the code:

```
function foo(data1, data2)  
    B(data1)  
    return A(data2)
```

where `data1` and `data2` are parameters. A compiler might translate to the following pseudo assembly code:

```
foo:  
    mov reg, [sp+data1] ; fetch data1 from stack (sp) parameter into a scratch register.  
    push reg            ; put data1 on stack where B expects it  
    call B              ; B uses data1  
    pop                ; remove data1 from stack  
    mov reg, [sp+data2] ; fetch data2 from stack (sp) parameter into a scratch register.  
    push reg            ; put data2 on stack where A expects it  
    call A              ; A uses data2  
    pop                ; remove data2 from stack.  
    ret
```

A tail call optimizer could then change the code to:

```

foo:
    mov reg, [sp+data1] ; fetch data1 from stack (sp) parameter into a scratch register.
    push reg             ; put data1 on stack where B expects it
    call B               ; B uses data1
    pop                 ; remove data1 from stack
    mov reg, [sp+data2] ; fetch data2 from stack (sp) parameter into a scratch register.
    mov [sp+data2], reg ; put data2 where A expects it
    jmp A               ; A uses data2 and returns immediately to caller.

```

This changed code is more efficient both in terms of execution speed and use of stack space.

Through trampolining

However, since many Scheme compilers use C as an intermediate target code, the problem comes down to coding tail recursion in C without growing the stack, even if the back-end compiler does not optimize tail calls. Many implementations achieve this by using a device known as a trampoline, a piece of code that repeatedly calls functions. All functions are entered via the trampoline. When a function has to call another, instead of calling it directly it returns the address of the function to be called, the arguments to be used, and so on, to the trampoline. This ensures that the C stack does not grow and iteration can continue indefinitely.

It is possible to implement trampolining using higher-order functions in languages that support them, such as Groovy, Visual Basic .NET and C#. ^[9]

Using a trampoline for all function calls is rather more expensive than the normal C function call, so at least one Scheme compiler, Chicken, uses a technique first described by Henry Baker from an unpublished suggestion by Andrew Appel,^[10] in which normal C calls are used but the stack size is checked before every call. When the stack reaches its maximum permitted size, objects on the stack are garbage-collected using the Cheney algorithm by moving all live data into a separate heap. Following this, the stack is unwound ("popped") and the program resumes from the state saved just before the garbage collection. Baker says "Appel's method avoids making a large number of small trampoline bounces by occasionally jumping off the Empire State Building."^[10] The garbage collection ensures that mutual tail recursion can continue indefinitely. However, this approach requires that no C function call ever returns, since there is no guarantee that its caller's stack frame still exists; therefore, it involves a much more dramatic internal rewriting of the program code: continuation-passing style.

References

- [1] <http://cstheory.stackexchange.com/q/7540/1037>
- [2] http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-8.html#node_sec_5.11
- [3] http://www.r6rs.org/final/html/r6rs-rationale/r6rs-rationale-Z-H-7.html#node_sec_5.3
- [4] http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-14.html#node_sec_11.20
- [5] D. H. D. Warren, *DAI Research Report 141*, University of Edinburgh, 1980.
- [6] Guy Lewis Steele, Jr.. "Debunking the 'Expensive Procedure Call' Myth, or, Procedure Call Implementations Considered Harmful, or, Lambda: The Ultimate GOTO". MIT AI Lab. AI Lab Memo AIM-443. October 1977.
- [7] R5RS Sec. 3.5, Richard Kelsey, William Clinger, Jonathan Rees et al. (August 1998). "Revised⁵ Report on the Algorithmic Language Scheme" (<http://www.schemers.org/Documents/Standards/R5RS/>). *Higher-Order and Symbolic Computation* **11** (1): 7–105. doi:10.1023/A:1010051815785..
- [8] <http://perldoc.perl.org/functions/goto.html>
- [9] Samuel Jack, Bouncing on your tail (<http://blog.functionalfun.net/2008/04/bouncing-on-your-tail.html>). *Functional Fun*. April 9, 2008.
- [10] Henry Baker, "CONS Should Not CONS Its Arguments, Part II: Cheney on the M.T.A." (<http://home.pipeline.com/~hbaker1/CheneyMTA.html>)

This article is based on material taken from the Free On-line Dictionary of Computing prior to 1 November 2008 and incorporated under the "relicensing" terms of the GFDL, version 1.3 or later.

Lazy evaluation

In programming language theory, **lazy evaluation** or **call-by-need**^[1] is an evaluation strategy which delays the evaluation of an expression until its value is needed (non-strict evaluation) and which also avoids repeated evaluations (sharing).^{[2][3]} The sharing can reduce the running time of certain functions by an exponential factor over other non-strict evaluation strategies, such as call-by-name.

The benefits of lazy evaluation include:

- Performance increases by avoiding needless calculations, and error conditions in evaluating compound expressions
- The ability to construct potentially infinite data structures
- The ability to define control flow (structures) as abstractions instead of primitives

Lazy evaluation can lead to reduction in memory footprint, since values are created when needed.^[4] However, with lazy evaluation, it is difficult to combine with imperative features such as exception handling and input/output, because the order of operations becomes indeterminate. Lazy evaluation can introduce space leaks.^[5]

The opposite of lazy actions is eager evaluation, sometimes known as strict evaluation. Eager evaluation is commonly believed as the default behavior used in programming languages.

History

Lazy evaluation was introduced for the lambda calculus by (Wadsworth 1971) and for programming languages independently by (Henderson & Morris 1976) and (Friedman & Wise 1976).^[6]

Applications

Delayed evaluation is used particularly in functional programming languages. When using delayed evaluation, an expression is not evaluated as soon as it gets bound to a variable, but when the evaluator is forced to produce the expression's value. That is, a statement such as `x:=expression;` (i.e. the assignment of the result of an expression to a variable) clearly calls for the expression to be evaluated and the result placed in `x`, but what actually is in `x` is irrelevant until there is a need for its value via a reference to `x` in some later expression whose evaluation could itself be deferred, though eventually the rapidly growing tree of dependencies would be pruned to produce some symbol rather than another for the outside world to see.^[7]

Some programming languages delay evaluation of expressions by default, and some others provide functions or special syntax to delay evaluation. In Miranda and Haskell, evaluation of function arguments is delayed by default. In many other languages, evaluation can be delayed by explicitly suspending the computation using special syntax (as with Scheme's "delay" and "force" and OCaml's "lazy" and "Lazy.force") or, more generally, by wrapping the expression in a thunk. The object representing such an explicitly delayed evaluation is called a future or promise. Perl 6 uses lazy evaluation of lists, so one can assign infinite lists to variables and use them as arguments to functions, but unlike Haskell and Miranda, Perl 6 doesn't use lazy evaluation of arithmetic operators and functions by default.^[7]

Delayed evaluation has the advantage of being able to create calculable infinite lists without infinite loops or size matters interfering in computation. For example, one could create a function that creates an infinite list (often called a *stream*) of Fibonacci numbers. The calculation of the n -th Fibonacci number would be merely the extraction of that element from the infinite list, forcing the evaluation of only the first n members of the list.^{[8][9]}

For example, in Haskell, the list of all Fibonacci numbers can be written as:^[9]

```
fibs = 0 : 1 : zipWith (+) fibs (tail fibs)
```

In Haskell syntax, ":" prepends an element to a list, `tail` returns a list without its first element, and `zipWith` uses a specified function (in this case addition) to combine corresponding elements of two lists to produce a third.^[8]

Provided the programmer is careful, only the values that are required to produce a particular result are evaluated. However, certain calculations may result in the program attempting to evaluate an infinite number of elements; for example, requesting the length of the list or trying to sum the elements of the list with a fold operation would result in the program either failing to terminate or running out of memory.

Control structures

In most eager languages, *if* statements evaluate in a lazy fashion.

```
if a then b else c
```

evaluates (a), then if and only if (a) evaluates to true does it evaluate (b), otherwise it evaluates (c). That is, either (b) or (c) will not be evaluated. Conversely, in an eager language the expected behavior is that

```
define f(x,y) = 2*x
set k = f(e,5)
```

will still evaluate (e) and (f) when computing (k). However, user-defined control structures depend on exact syntax, so for example

```
define g(a,b,c) = if a then b else c
l = g(h,i,j)
```

(i) and (j) would both be evaluated in an eager language. While in

```
l' = if h then i else j
```

(i) or (j) would be evaluated, but never both.

Lazy evaluation allows control structures to be defined normally, and not as primitives or compile-time techniques. If (i) or (j) have side effects or introduce run time errors, the subtle differences between (l) and (l') can be complex. As most programming languages are Turing-complete, it is possible to introduce user-defined lazy control structures in eager languages as functions, though they may depart from the language's syntax for eager evaluation: Often the involved code bodies (like (i) and (j)) need to be wrapped in a function value, so that they are executed only when called.

Short-circuit evaluation of Boolean control structures is sometimes called *lazy*.

Working with infinite data structures

Many languages offer the notion of *infinite data-structures*. These allow definitions of data to be given in terms of infinite ranges, or unending recursion, but the actual values are only computed when needed. Take for example this trivial program in Haskell:

```
numberFromInfiniteList :: Int -> Int
numberFromInfiniteList n = infinity !! n - 1
  where infinity = [1..]

main = print $ numberFromInfiniteList 4
```

In the function `numberFromInfiniteList`, the value of `infinity` is an infinite range, but until an actual value (or more specifically, a specific value at a certain index) is needed, the list is not evaluated, and even then it is only evaluated as needed (that is, until the desired index.)

Other uses

In computer windowing systems, the painting of information to the screen is driven by *expose events* which drive the display code at the last possible moment. By doing this, they avoid computating needless display content.^[10]

Another example of laziness in modern computer systems is copy-on-write page allocation or demand paging, where memory is allocated only when a value stored in that memory is changed.^[10]

Laziness can be useful for high performance scenarios. An example is the Unix mmap function, which provides *demand driven* loading of pages from disk, so that only those pages actually touched are loaded into memory, and unneeded memory is not allocated.

Laziness in eager languages

Python

In Python 2.x the `range()` function^[11] computes a list of integers (eager or immediate evaluation):

```
>>> r = range(10)
>>> print r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print r[3]
3
```

In Python 3.x the `range()` function^[12] returns an iterator which computes elements of the list on demand (lazy or deferred evaluation):

```
>>> r = range(10)
>>> print(r)
range(0, 10)
>>> print(r[3])
3
```

This change to lazy evaluation saves execution time for large ranges which may never be fully referenced and memory usage for large ranges where only one or a few elements are needed at any time.

Python manifests lazy evaluation by implementing iterators (lazy sequences) unlike tuple or list sequences. For instance:

```
>>> list = range(10)
>>> iterator = iter(list)
>>> print list
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print iterator
<listiterator object at 0xf7e8dd4c>
>>> print iterator.next()
0
```

The above example shows that lists are evaluated when called, but in case of iterator, the first element '0' is printed when need arises.

Controlling eagerness in lazy languages

In lazy programming languages such as Haskell, although the default is to evaluate expressions only when they are demanded, it is possible in some cases to make code more eager—or conversely, to make it more lazy again after it has been made more eager. This can be done by explicitly coding something which forces evaluation (which may make the code more eager) or avoiding such code (which may make the code more lazy). *Strict* evaluation usually implies eagerness, but they are technically different concepts.

However, there is an optimisation implemented in some compilers called strictness analysis, which, in some cases, allows the compiler to infer that a value will always be used. In such cases, this may render the programmer's choice of whether to force that particular value or not, irrelevant, because strictness analysis will force strict evaluation.

In Haskell, marking constructor fields strict means that their values will always be demanded immediately. The `seq` function can also be used to demand a value immediately and then pass it on, which is useful if a constructor field should generally be lazy. However, neither of these techniques implements *recursive* strictness—for that, a function called `deepSeq` was invented.

Also, pattern matching in Haskell 98 is strict by default, so the `~` qualifier has to be used to make it lazy.

Notes

- [1] Hudak 1989, p. 384
- [2] David Anthony Watt; William Findlay (2004). *Programming language design concepts* (<http://books.google.com/books?id=vogP3P2L4tgC&pg=PA367>). John Wiley and Sons. pp. 367–368. ISBN 978-0-470-85320-7. . Retrieved 30 December 2010.
- [3] Reynolds 1998, p. 307
- [4] Chris Smith (22 October 2009). *Programming F#* (<http://books.google.com/books?id=gzVdyw2WoXMC&pg=PA79>). O'Reilly Media, Inc.. p. 79. ISBN 978-0-596-15364-9. . Retrieved 31 December 2010.
- [5] Launchbury 1993
- [6] Reynolds 1998, p. 312
- [7] Philip Wadler (2006). *Functional and logic programming: 8th international symposium, FLOPS 2006, Fuji-Susono, Japan, April 24-26, 2006 : proceedings* (<http://books.google.com/books?id=gZzLFFZfc1sC&pg=PA149>). Springer. p. 149. ISBN 978-3-540-33438-5. . Retrieved 14 January 2011.
- [8] Daniel Le Métayer (2002). *Programming languages and systems: 11th European Symposium on Programming, ESOP 2002, held as part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2002, Grenoble, France, April 8-12, 2002 : proceedings* (<http://books.google.com/books?id=dYZyzp-I9hQC&pg=PA129>). Springer. pp. 129–132. ISBN 978-3-540-43363-7. . Retrieved 14 January 2011.
- [9] Association for Computing Machinery; ACM Special Interest Group on Programming Languages (1 January 2002). *Proceedings of the 2002 ACM SIGPLAN Haskell Workshop (Haskell '02): Pittsburgh, Pennsylvania, USA ; October 3, 2002* (<http://books.google.com/books?id=hsBQAAAAMAAJ>). Association for Computing Machinery. p. 40. ISBN 978-1-58113-605-0. . Retrieved 14 January 2011.
- [10] Lazy and Speculative Execution (<http://research.microsoft.com/en-us/um/people/blampson/slides/lazyandspeculative.ppt>) Butler Lampson Microsoft Research OPODIS, Bordeaux, France 12 December 2006
- [11] <http://docs.python.org/library/functions.html#range>
- [12] <http://docs.python.org/py3k/library/functions.html#range>

References

- Hudak, Paul (September 1989). "Conception, Evolution, and Application of Functional Programming Languages" (<http://portal.acm.org/citation.cfm?id=72554>). *ACM Computing Surveys* **21** (3): 383–385.
- Reynolds, John C. (1998). *Theories of programming languages* (<http://books.google.com/books?id=Hkl01IHJMcQC&pg=PA307>). Cambridge University Press. ISBN 978052159414 .

Further reading

- Wadsworth, Christopher P. (1971). *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, Oxford University
- Henderson, Peter; Morris, James H. (January 1976). "A Lazy Evaluator" (<http://portal.acm.org/citation.cfm?id=811543>). *Conference Record of the Third ACM symposium on Principles of Programming Languages*.

- Friedman, D. P.; Wise, David S. (1976). S. Michaelson and R. Milner, ed. "Cons should not evaluate its arguments" (<http://www.cs.indiana.edu/pub/techreports/TR44.pdf>). *Automata Languages and Programming Third International Colloquium* (Edinburgh University Press).
- Launchbury, John (1993). "A Natural Semantics for Lazy Evaluation" (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.2016>). *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL '93)*. doi:10.1145/158511.158618.

Design patterns

- John Hughes. "Why functional programming matters" (<http://www.cs.kent.ac.uk/people/staff/dat/miranda/whyfp90.pdf>). *The Computer Journal* - Special issue on lazy functional programming (<http://comjnl.oxfordjournals.org/content/32/2.toc>). Volume 32 Issue 2, April 1989.
- Philip Wadler. "How to replace failure by a list of successes a method for exception handling, backtracking, and pattern matching in lazy functional languages" (<http://www.springerlink.com/content/y7450255v2670167/>). *Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, 1985, Volume 201/1985, 113-128.

Laziness in strict languages

- Philip Wadler, Walid Taha, and David MacQueen. "How to add laziness to a strict language, without even being odd" (<http://homepages.inf.ed.ac.uk/wadler/papers/lazyinstrict/lazyinstrict.ps.gz>). Workshop on Standard ML, Baltimore, September 1998.

Blog posts by computer scientists

- Robert Harper. "The Point of Laziness" (<http://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/>)
- Lennart Augustsson. "More points for lazy evaluation" (<http://augustss.blogspot.com/2011/05/more-points-for-lazy-evaluation-in.html>)

External links

- Lazy Evaluation (<http://c2.com/cgi/wiki?LazyEvaluation>) at the Portland Pattern Repository
- Lazy evaluation (http://haskell.org/haskellwiki/Haskell/Lazy_evaluation) at Haskell Wiki
- Functional programming in Python becomes lazy (http://gnosis.cx/publish/programming/charming_python_b13.html)
- Lazy function argument evaluation (<http://www.digitalmars.com/d/lazy-evaluation.html>) in the D language
- Lazy evaluation macros (http://nemerle.org/Lazy_evaluation) in Nemerle
- Lazy programming and lazy evaluation (<http://www-128.ibm.com/developerworks/linux/library/l-lazyprog.html>) in Scheme
- Lambda calculus in Boost Libraries (http://spirit.sourceforge.net/dl_docs/phoenix-2/libs/spirit/phoenix/doc/html/phoenix/introduction.html) in C++ language
- Lazy Evaluation (<http://perldesignpatterns.com/?LazyEvaluation>) in Perl

Haskell (programming language)

Haskell



Paradigm(s)	functional, lazy/non-strict, modular
Appeared in	1990
Designed by	Simon Peyton Jones, Lennart Augustsson, Dave Barton, Brian Boutel, Warren Burton, Joseph Fasel, Kevin Hammond, Ralf Hinze, Paul Hudak, John Hughes, Thomas Johnsson, Mark Jones, John Launchbury, Erik Meijer, John Peterson, Alastair Reid, Colin Runciman, Philip Wadler
Stable release	Haskell 2010 ^[1] (July 2010)
Preview release	to be announced as Haskell 2012 ^[2]
Typing discipline	static, strong, inferred
Major implementations	GHC, Hugs, NHC ^[3] , JHC ^[4] , Yhc, UHC ^[5]
Dialects	Helium, Gofer, Hugs, Ω mega
Influenced by	APL, Clean, ^[6] FP, ^[6] Gofer, ^[6] Hope and Hope ⁺ , ^[6] Id, ^[6] ISWIM, ^[6] KRC, ^[6] Lisp, ^[6] Miranda, ^[6] ML and Standard ML, ^[6] Orwell, SASL, ^[6] SISAL, ^[6] Scheme ^[6]
Influenced	Agda, ^[7] Bluespec, C++11/Concepts, ^[8] C#/LINQ, ^{[9][10][11][12]} CAL, Cayenne, ^[9] Clean, ^[9] Clojure, ^[13] CoffeeScript, ^[14] Curry, ^[9] Epigram, Escher, F#, ^[15] Isabelle, ^[9] Java/Generics, ^[9] Kaya, Mercury, ^[9] Omega, Perl 6, ^[16] Python, ^{[9][17]} Qi, Scala, ^{[9][18]} Timber, Visual Basic 9.0 ^{[9][10]}
OS	Cross-platform
Usual filename extensions	.hs, .lhs
Website	haskell.org ^[19]

Haskell (/hæskəl/)^[20] is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing.^[21] It is named after logician Haskell Curry.^[22] In Haskell, "a function is a first-class citizen" of the programming language.^[23] As a functional programming language, the primary control construct is the function.

History

Following the release of Miranda by Research Software Ltd, in 1985, interest in lazy functional languages grew: by 1987, more than a dozen non-strict, purely functional programming languages existed. Of these, Miranda was the most widely used, but was not in the public domain. At the conference on Functional Programming Languages and Computer Architecture (FPCA '87) in Portland, Oregon, a meeting was held during which participants formed a strong consensus that a committee should be formed to define an open standard for such languages. The committee's purpose was to consolidate the existing functional languages into a common one that would serve as a basis for future research in functional-language design.^[24]

Haskell 1.0 to 1.4

The first version of Haskell ("Haskell 1.0") was defined in 1990.^[22] The committee's efforts resulted in a series of language definitions (1.0, 1.1, 1.2, 1.3, 1.4).

Haskell 98

In late 1997, the series culminated in **Haskell 98**, intended to specify a stable, minimal, portable version of the language and an accompanying standard library for teaching, and as a base for future extensions. The committee expressly welcomed the creation of extensions and variants of Haskell 98 via adding and incorporating experimental features.^[24]

In February 1999, the Haskell 98 language standard was originally published as "The Haskell 98 Report".^[24] In January 2003, a revised version was published as "Haskell 98 Language and Libraries: The Revised Report".^[21] The language continues to evolve rapidly, with the Glasgow Haskell Compiler (GHC) implementation representing the current *de facto* standard.

Haskell Prime

In early 2006, the process of defining a successor to the Haskell 98 standard, informally named **Haskell Prime**, was begun.^[25] This is an ongoing incremental process to revise the language definition, producing a new revision once per year. The first revision, named **Haskell 2010**, was announced in November 2009^[1] and published in July 2010.

Haskell 2010

Haskell 2010 adds the Foreign Function Interface (FFI) to Haskell, allowing for bindings to other programming languages, fixes some syntax issues (changes in the formal grammar) and bans so-called "n-plus-k-patterns", that is, definitions of the form `fact (n+1) = (n+1) * fact n` are no longer allowed. It introduces the Language-Pragma-Syntax-Extension which allows for designating a Haskell source as Haskell 2010 or requiring certain Extensions to the Haskell Language. The names of the extensions introduced in Haskell 2010 are `DoAndIfThenElse`, `HierarchicalModules`, `EmptyDataDeclarations`, `FixityResolution`, `ForeignFunctionInterface`, `LineCommentSyntax`, `PatternGuards`, `RelaxedDependencyAnalysis`, `LanguagePragma`, `NoNPlusKPatterns`.^[1]

Features

Haskell features lazy evaluation, pattern matching, list comprehension, type classes, and type polymorphism. It is a purely functional language, which means that in general, functions in Haskell do not have side effects. There is a distinct type for representing side effects, orthogonal to the type of functions. A pure function may return a side effect which is subsequently executed, modeling the impure functions of other languages.

Haskell has a strong, static type system based on Hindley–Milner type inference. Haskell's principal innovation in this area is to add type classes, which were originally conceived as a principled way to add overloading to the language,^[26] but have since found many more uses.^[27]

The type which represents side effects is an example of a monad. Monads are a general framework which can model different kinds of computation, including error handling, nondeterminism, parsing, and software transactional memory. Monads are defined as ordinary datatypes, but Haskell provides some syntactic sugar for their use.

The language has an open, published specification,^[21] and multiple implementations exist.

There is an active community around the language, and more than 3600 third-party open-source libraries and tools are available in the online package repository Hackage^{[28][29]}.

The main implementation of Haskell, GHC, is both an interpreter and native-code compiler that runs on most platforms. GHC is noted for its high-performance implementation of concurrency and parallelism,^[30] and for having a rich type system incorporating recent innovations such as generalized algebraic data types and Type Families^[31].

Code examples

The following is a Hello world program written in Haskell (note that all but the last line can be omitted):

```
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Here is the factorial function in Haskell, defined in a few different ways:

```
-- Type annotation (optional)
factorial :: Integer -> Integer

-- Using recursion
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- Using recursion but written without pattern matching
factorial n = if n > 0 then n * factorial (n-1) else 1

-- Using a list
factorial n = product [1..n]

-- Using fold (implements product)
factorial n = foldl1 (*) [1..n]

-- Point-free style
factorial = foldr (*) 1 . enumFromTo 1
```

An efficient implementation of the Fibonacci numbers, as an infinite list, is this:

```
-- Type annotation (optional)
fib :: Int -> Integer

-- With self-referencing data
fib n = fibs !! n
      where fibs = 0 : scanl (+) 1 fibs
            -- 0,1,1,2,3,5, ...

-- Same, coded directly
fib n = fibs !! n
      where fibs = 0 : 1 : next fibs
            next (a : t@(b:_)) = (a+b) : next t

-- Similar idea, using zipWith
fib n = fibs !! n
      where fibs = 0 : 1 : zipWith (+) fibs (tail fibs)

-- Using a generator function
```

```
fib n = fibs (0,1) !! n
    where fibs (a,b) = a : fibs (b,a+b)
```

The "Int" type refers to a machine-sized integer (used as a list subscript with the `!!` operator), while "Integer" is an arbitrary-precision integer. For example, the above code quickly computes "fib 10000" as a 2090-digit number.

Implementations

All listed implementations are distributed under open source licenses. There are currently no proprietary Haskell implementations.

The following implementations comply fully, or very nearly, with the Haskell 98 standard.

- The **Glasgow Haskell Compiler** (GHC) compiles to native code on a number of different architectures—as well as to ANSI C—using C++ as an intermediate language. GHC is probably the most popular Haskell compiler, and there are quite a few useful libraries (e.g. bindings to OpenGL) that will work only with GHC. GHC is also distributed along with the Haskell platform.
- **HBC**^[32] is another native-code Haskell compiler. It has not been actively developed for some time but is still usable.
- The **Utrecht Haskell Compiler**^[5] (UHC) is a Haskell implementation from Utrecht University. UHC supports almost all Haskell 98 features plus many experimental extensions. It is implemented using attribute grammars and is currently mainly used for research into generated type systems and language extensions.
- **Hugs, the Haskell User's Gofer System**, is a bytecode interpreter. It offers fast compilation of programs and reasonable execution speed. It also comes with a simple graphics library. Hugs is good for people learning the basics of Haskell, but is by no means a "toy" implementation. It is the most portable and lightweight of the Haskell implementations.
- **Jhc**^[4] is a Haskell compiler written by John Meacham emphasising speed and efficiency of generated programs as well as exploration of new program transformations. LHC is a recent fork of Jhc.
- **nhc98**^[3] is another bytecode compiler, but the bytecode runs significantly faster than with Hugs. Nhc98 focuses on minimizing memory usage, and is a particularly good choice for older, slower machines.
- **Yhc, the York Haskell Compiler** was a fork of nhc98, with the goals of being simpler, more portable and more efficient, and integrating support for Hat^[33], the Haskell tracer. It also featured a JavaScript backend allowing users to run Haskell programs in a web browser^[34].

Implementations below are not fully Haskell 98 compliant, and use a language that is a variant of Haskell:

- **Gofer** was an educational dialect of Haskell, with a feature called "constructor classes", developed by Mark Jones. It was supplanted by Hugs (see above).
- **Helium** is a newer dialect of Haskell. The focus is on making it easy to learn by providing clearer error messages. It currently lacks full support for type classes, rendering it incompatible with many Haskell programs.

Applications

Haskell is increasingly being used in commercial situations.^[35] Audrey Tang's Pugs is an implementation for the long-forthcoming Perl 6 language with an interpreter and compilers that proved useful after just a few months of its writing; similarly, GHC is often a testbed for advanced functional programming features and optimizations. Darcs is a revision control system written in Haskell, with several innovative features. Linspire GNU/Linux chose Haskell for system tools development.^[36] Xmonad is a window manager for the X Window System, written entirely in Haskell.

Bluespec SystemVerilog is a language for semiconductor design that is an extension of Haskell. Additionally, Bluespec, Inc.'s tools are implemented in Haskell. Cryptol, a language and toolchain for developing and verifying cryptographic algorithms, is implemented in Haskell. Notably, the first formally verified microkernel, seL4 was verified using Haskell.

There are also Web frameworks increasingly productive^[37], among them:

- Yesod
- Happstack
- Snap^[38]

Related languages

Concurrent Clean is a close relative of Haskell. Its biggest deviation from Haskell is in the use of uniqueness types instead of monads for I/O and side-effects.

A series of languages inspired by Haskell, but with different type systems, have been developed, including:

- Epigram, a functional language with dependent types suitable for proving properties of programs
- Agda, a functional language with dependent types

JVM-based:

- Frege, a Haskell-like language with Java's scalar types and good Java integration.^{[39][40][41]}
- Jaskell, a functional scripting programming language that runs in Java VM.^[42]

Other related languages include:

- Curry, a language based on Haskell

Haskell has served as a testbed for many new ideas in language design. There have been a wide number of Haskell variants produced, exploring new language ideas, including:

• Parallel Haskell:

- From Glasgow University, supports clusters of machines or single multiprocessors.^{[43][44]} Also within Haskell is support for Symmetric Multiprocessor parallelism.^[45]
- From MIT^[46]
- Distributed Haskell (formerly Goffin) and Eden.
- Eager Haskell^[47], based on speculative evaluation.
- Several object-oriented versions: Haskell++, and Mondrian.
- Generic Haskell, a version of Haskell with type system support for generic programming.
- O'Haskell, an extension of Haskell adding object-orientation and concurrent programming support which "has reportedly been superseded by Timber."^[48]
- Disciple^[49], a strict-by-default (laziness available by annotation) dialect of Haskell which supports destructive update, computational effects, type directed field projections and allied functional goodness.
- Scotch, a kind of hybrid of Haskell and Python^[50]
- Hume, a strict functional programming language for embedded systems based on processes as stateless automata over a sort of tuples of single element mailbox channels where the state is kept by feedback into the mailboxes, and a mapping description from outputs to channels as box wiring, with a Haskell-like expression language and syntax.

Criticism

Jan-Willem Maessen, in 2002, and Simon Peyton Jones, in 2003, discussed problems associated with lazy evaluation while also acknowledging the theoretical motivation for it,^{[51][52]} in addition to purely practical considerations such as improved performance.^[53] They note that, in addition to adding some performance overhead, lazy evaluation makes it more difficult for programmers to reason about the performance of their code (particularly its space usage).

Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn in 2003 also observed some stumbling blocks for Haskell learners: "The subtle syntax and sophisticated type system of Haskell are a double edged sword — highly appreciated by experienced programmers but also a source of frustration among beginners, since the generality of Haskell often leads to cryptic error messages."^[54] To address these, researchers from Utrecht University developed an advanced interpreter called Helium which improved the user-friendliness of error messages by limiting the generality of some Haskell features, and in particular removing support for type classes.

Ben Lippmeier designed Disciple^[55] as a strict-by-default (lazy by explicit annotation) dialect of Haskell with a type-and-effect system, to address Haskell's difficulties in reasoning about lazy evaluation and in using traditional data structures such as mutable arrays.^[56] He argues (p. 20) that "destructive update furnishes the programmer with two important and powerful tools... a set of efficient array-like data structures for managing collections of objects, and ... the ability to broadcast a new value to all parts of a program with minimal burden on the programmer."

Robert Harper, one of the authors of Standard ML, has given his reasons for not using Haskell to teach introductory programming. Among these are the difficulty of reasoning about resource usage with non-strict evaluation, that lazy evaluation complicates the definition of data types and inductive reasoning,^[57] and the "inferiority" of Haskell's class system compared to ML's module system.^[58]

Conferences and workshops

The Haskell community meets regularly for research and development activities. The primary events are:

- The Haskell Symposium^[59] (formerly the Haskell Workshop)
- The Haskell Implementors Workshop^[60]
- The International Conference on Functional Programming

Since 2006, there have been a series of organized "hackathons", the Hac^[61] series, aimed at improving the programming language tools and libraries.^[62]

Since 2005, a growing number of Haskell User Groups^[63] have formed, in the United States, Canada, Australia, South America, Europe and Asia.

References

- [1] Marlow, Simon (24 November 2009). "Announcing Haskell 2010" (<http://www.haskell.org/pipermail/haskell/2009-November/021750.html>). *Haskell mailing list*. . Retrieved 12 March 2011.
- [2] (<http://www.haskell.org/pipermail/haskell/2011-January/022497.html>)
- [3] <http://www.cs.york.ac.uk/fp/nhc98/>
- [4] <http://reptae.net/john/computer/jhc/>
- [5] <http://www.cs.uu.nl/wiki/UHC>
- [6] Peyton Jones 2003, p. xi
- [7] Norell, Ulf (2008). "Dependently Typed Programming in Agda" (<http://www.cse.chalmers.se/~ulfn/papers/afp08/tutorial.pdf>). Gothenburg: Chalmers University. . Retrieved 9 February 2012.
- [8] Stroustrup, Bjarne; Sutton, Andrew (2011). *Design of Concept Libraries for C++* (<http://www2.research.att.com/~bs/sle2011-concepts.pdf>). .
- [9] Hudak et al. 2007, pp. 12-45-46
- [10] Meijer, Erik. "Confessions of a Used Programming Language Salesman: Getting the Masses Hooked on Haskell" (<http://research.microsoft.com/en-us/um/people/emeijer/papers/es012-meijer.pdf>). *OOPSLA 2007*. .
- [11] Meijer, Erik (1 October 2009). "C9 Lectures: Dr. Erik Meijer - Functional Programming Fundamentals, Chapter 1 of 13" (<http://channel9.msdn.com/shows/Going+Deep/Lecture-Series-Erik-Meijer-Functional-Programming-Fundamentals-Chapter-1/>). *Channel 9*. Microsoft. .

- Retrieved 9 February 2012.
- [12] Drobis, Sadek (4 March 2009). "Erik Meijer on LINQ" (<http://www.infoq.com/interviews/LINQ-Erik-Meijer>). *InfoQ* (QCon SF 2008: C4Media Inc.). . Retrieved 9 February 2012.
 - [13] Hickey, Rich. "Clojure Bookshelf" (<http://www.amazon.com/gp/richpub/listmania/fullview/R3LG3ZBZS4GCTH>). *Listmania!*. Amazon.com. . Retrieved 9 February 2012.
 - [14] Heller, Martin (18 October 2011). "Turn up your nose at Dart and smell the CoffeeScript" (<http://www.javaworld.com/javaworld/jw-10-2011/111018-coffeescript-vs-dart.html>). *JavaWorld* (InfoWorld). . Retrieved 9 February 2012.
 - [15] Syme, Don; Granicz, Adam; Cistermino, Antonio (2007). *Expert F#*. Apress. p. 2. "F# also draws from Haskell particularly with regard to two advanced language features called *sequence expressions* and *workflows*."
 - [16] "Glossary of Terms and Jargon" (http://www.perlfoundation.org/perl6/index.cgi?glossary_of_terms_and_jargon). *Perl Foundation Perl 6 Wiki*. The Perl Foundation. 28 February. . Retrieved 9 February 2012.
 - [17] Kuchling, A. M.. "Functional Programming HOWTO" (<http://docs.python.org/howto/functional.html>). *Python v2.7.2 documentation*. Python Software Foundation. . Retrieved 9 February 2012.
 - [18] Fogus, Michael (6 August 2010). "MartinOdersky take(5) toList" (<http://blog.fogus.me/2010/08/06/martinodersky-take5-tolist/>). *Send More Paramedics*. . Retrieved 9 February 2012.
 - [19] <http://haskell.org>
 - [20] Chevalier, Tim (28 January 2008). "anybody can tell me the pronunciation of "haskell"?" (<http://www.haskell.org/pipermail/haskell-cafe/2008-January/038756.html>). *Haskell-cafe mailing list*. . Retrieved 12 March 2011.
 - [21] Peyton Jones 2003
 - [22] Hudak et al. 2007
 - [23] Burstall, Rod (2000). "Christopher Strachey—Understanding Programming Languages". *Higher-Order and Symbolic Computation* **13** (52).
 - [24] Peyton Jones 2003, Preface
 - [25] "Welcome to Haskell" (<http://hackage.haskell.org/trac/haskell-prime>). *The Haskell' Wiki*. .
 - [26] Wadler, P.; Blott, S. (1989). "How to make ad-hoc polymorphism less ad hoc". *Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (ACM): 60–76. doi:10.1145/75277.75283. ISBN 0-89791-294-2.
 - [27] Hallgren, T. (January 2001). "Fun with Functional Dependencies, or Types as Values in Static Computations in Haskell" (<http://www.cs.chalmers.se/~hallgren/Papers/wm01.html>). *Proceedings of the Joint CS/CE Winter Meeting* (Varberg, Sweden). .
 - [28] <http://hackage.haskell.org/packages/hackage.html>
 - [29] <http://hackage.haskell.org/cgi-bin/hackage-scripts/stats>
 - [30] Computer Language Benchmarks Game (<http://shootout.alioth.debian.org/>)
 - [31] http://www.haskell.org/ghc/docs/latest/html/users_guide/type-families.html
 - [32] <http://www.cs.chalmers.se/~augustss/hbc/hbc.html>
 - [33] <http://www.haskell.org/hat/>
 - [34] http://haskell.org/haskellwiki/Haskell_in_web_browser
 - [35] See Industrial Haskell Group (<http://industry.haskell.org/index>) for collaborative development, Commercial Users of Functional Programming (<http://cufp.galois.com/>) for specific projects and Haskell in industry (http://www.haskell.org/haskellwiki/Haskell_in_industry) for a list of companies using Haskell commercially
 - [36] "Linspire/Freespire Core OS Team and Haskell" (<http://urchin.earth.li/pipermail/debian-haskell/2006-May/000169.html>). *Debian Haskell mailing list*. May 2006. .
 - [37] HaskellWiki - Haskell web frameworks (<http://www.haskell.org/haskellwiki/Web/Frameworks>)
 - [38] (<http://snapframework.com/>)
 - [39] The Frege prog. lang. (<http://fregepl.blogspot.com>)
 - [40] Project Frege at google code (<http://code.google.com/p/frege/>)
 - [41] Hello World and more with Frege (<http://mmhelloworld.blogspot.com.es/2012/02/hello-world-frege.html>)
 - [42] Jaskell (<http://jaskell.codehaus.org/>)
 - [43] Glasgow Parallel Haskell (<http://www.macs.hw.ac.uk/~dsg/gph/>)
 - [44] GHC Language Features: Parallel Haskell (http://www.haskell.org/ghc/docs/6.6/html/users_guide/lang-parallel.html)
 - [45] Using GHC: Using SML parallelism (http://www.haskell.org/ghc/docs/6.6/html/users_guide/sec-using-smp.html)
 - [46] MIT Parallel Haskell (<http://csg.csail.mit.edu/projects/languages/ph.shtml>)
 - [47] <http://csg.csail.mit.edu/pubs/haskell.html>
 - [48] OHaskell at HaskellWiki (<http://www.haskell.org/haskellwiki/O'Haskell>)
 - [49] <http://disciple.ouroborus.net/>
 - [50] Scotch (<http://www.bendmorris.com/2011/01/what-problem-does-scotch-solve.html>)
 - [51] Jan-Willem Maessen. *Eager Haskell: Resource-bounded execution yields efficient iteration*. Proceedings of the 2002 ACM SIGPLAN workshop on Haskell.
 - [52] Simon Peyton Jones. *Wearing the hair shirt: a retrospective on Haskell* (<http://research.microsoft.com/~simonpj/papers/haskell-retrospective>). Invited talk at POPL 2003.
 - [53] Lazy evaluation can lead to excellent performance, such as in The Computer Language Benchmarks Game (<http://www.haskell.org/pipermail/haskell/2006-June/018127.html>)

- [54] Heeren, Bastiaan; Leijen, Daan; van IJzendoorn, Arjan (2003). "Helium, for learning Haskell" (<http://www.cs.uu.nl/~bastiaan/heeren-helium.pdf>). *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*. .
- [55] <http://www.haskell.org/haskellwiki/DDC>
- [56] Ben Lippmeier, Type Inference and Optimisation for an Impure World (<http://www.cse.unsw.edu.au/~benl/papers/thesis/lippmeier-impure-world.pdf>), Australian National University (2010) PhD thesis, chapter 1
- [57] Robert Harper. "The point of laziness" (<http://existentialtype.wordpress.com/2011/04/24/the-real-point-of-laziness/>). .
- [58] Robert Harper. "Modules matter most." (<http://existentialtype.wordpress.com/2011/04/16/modules-matter-most/>). .
- [59] <http://www.haskell.org/haskell-symposium/>
- [60] <http://haskell.org/haskellwiki/HaskellImplementorsWorkshop>
- [61] <http://haskell.org/haskellwiki/Hackathon>
- [62] "Hackathon - HaskellWiki" (<http://haskell.org/haskellwiki/Hackathon>). .
- [63] http://haskell.org/haskellwiki/User_groups

Further reading

Reports

- Peyton Jones, Simon, ed. (2003). *Haskell 98 Language and Libraries: The Revised Report* (<http://haskell.org/onlinereport/>). Cambridge University Press. ISBN 0521826144.

Textbooks

- Davie, Antony (1992). *An Introduction to Functional Programming Systems Using Haskell*. Cambridge University Press. ISBN 0-521-25830-8.
- Bird, Richard (1998). *Introduction to Functional Programming using Haskell* (<http://www.cs.ox.ac.uk/publications/books/functional/>) (2nd ed.). Prentice Hall Press. ISBN 0-13-484346-0.
- Thompson, Simon (1999). *Haskell: The Craft of Functional Programming* (<http://www.cs.kent.ac.uk/people/staff/sjt/craft2e/>) (2nd ed.). Addison-Wesley. ISBN 0-201-34275-8.
- Hudak, Paul (2000). *The Haskell School of Expression: Learning Functional Programming through Multimedia* (<http://www.cs.yale.edu/homes/hudak/SOE>). New York: Cambridge University Press. ISBN 0521643384.
- Hutton, Graham (2007). *Programming in Haskell* (<http://www.cs.nott.ac.uk/~gmh/book.html>). Cambridge University Press. ISBN 0521692695.
- Goerzen, John; O'Sullivan, Bryan (2008). *Real World Haskell* (<http://book.realworldhaskell.org>). Sebastopol: O'Reilly. ISBN 0-596-51498-0
- Lipováča, Miran (April 2011). *Learn You a Haskell for Great Good!* (<http://learnyouahaskell.com/>). San Francisco: No Starch Press. ISBN 978-1-59327-283-8.

History

- Hudak, Paul; Hughes, John; Peyton Jones, Simon; Wadler, Philip (2007). "A History of Haskell: Being Lazy with Class" (<http://research.microsoft.com/~simonpj/papers/history-of-haskell/history.pdf>). *Proceedings of the third ACM SIGPLAN conference on History of programming languages (HOPL III)*: 12-1–55.
doi:10.1145/1238844.1238856. ISBN 978-1-59593-766-7.

External links

- Official website (<http://haskell.org>)
- Language and library specification (http://www.haskell.org/haskellwiki/Language_and_library_specification) at the Haskell Wiki
- Haskell (<http://www.dmoz.org/Computers/Programming/Languages/Haskell/>) at the Open Directory Project

Tutorials

- Hudak, Paul; Peterson, John; Fasel, Joseph (June 2000). "A Gentle Introduction To Haskell, Version 98" (<http://haskell.org/tutorial/>). *Haskell.org*.
- Learn you a Haskell for great good! (<http://learnyouahaskell.com/>) by Miran Lipovača; assume no knowledge
- Try Haskell! (<http://tryhaskell.org/>), an in-browser interactive tutorial
- Yet Another Haskell Tutorial (<http://hal3.name/docs/daume02yaht.pdf>), by Hal Daumé III; assumes far less prior knowledge than official tutorial
- The Haskell Cheatsheet (<http://cheatsheet.codeslower.com/>), compact language reference and mini-tutorial

Various

- Yorgey, Brent (12 March 2009). "The Typeclassopedia" (<http://www.haskell.org/wikiupload/8/85/TMR-Issue13.pdf>). *The Monad.Reader* (13): 17–68
- Jones, William (5 August 2009). *Warp Speed Haskell* (<http://www.doc.ic.ac.uk/teaching/distinguished-projects/2009/w.jones.pdf>). Imperial College London.
- The Evolution of a Haskell Programmer (<http://www.willamette.edu/~fruehr/haskell/evolution.html>), slightly humorous overview of different programming styles available in Haskell
- Online Bibliography of Haskell Research (<http://haskell.readscheme.org/>)
- Haskell Weekly News (<http://contemplatecode.blogspot.com/search/label/HWN>)
- The Monad.Reader (<http://themonadreader.wordpress.com/>), quarterly magazine on Haskell topics
- Markus (29 August 2008). "Episode 108: Simon Peyton Jones on Functional Programming and Haskell" (<http://www.se-radio.net/2008/08/episode-108-simon-peyton-jones-on-functional-programming-and-haskell/>). *Software Engineering Radio* (Podcast).
- Leksah (<http://leksah.org/>), a GTK-based Haskell IDE written in Haskell
- Hamilton, Naomi (19 September 2008). "The A-Z of Programming Languages: Haskell" (http://www.computerworld.com.au/article/261007/a-z_programming_languages_haskell/). *Computerworld*.

Type system

<noinclude Mm q q q Vd XT9 </noinclude> A **type system** associates a *type* with each computed value. By examining the flow of these values, a type system attempts to ensure or prove that no *type errors* can occur. The particular type system in question determines exactly what constitutes a type error, but in general the aim is to prevent operations expecting a certain kind of value from being used with values for which that operation does not make sense (logic errors); memory errors will also be prevented. Type systems are often specified as part of programming languages, and built into the interpreters and compilers for them; although they can also be implemented as optional tools.

In computer science, a **type system** may be defined as "a tractable syntactic framework for classifying phrases according to the kinds of values they compute".^[1]

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

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

Fundamentals

Formally, type theory studies type systems. A programming language must have occurrence to type check using the *type system* whether at compiler time or runtime, manually annotated or automatically inferred. As Mark Manasse concisely put it:^[2]

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

Assigning a data type, what is called *typing*, gives meaning to a sequences of bits such as a value in memory or some object such as a variable. The hardware of a general purpose computer is unable to discriminate between for example a memory address and an instruction code, or between a character, an integer, or a floating-point number, because it makes no intrinsic distinction between any of the possible values of a sequence of bits might *mean*. Associating a sequence of bits with a type conveys that meaning to the programmable hardware to form a *symbolic system* composed of that hardware and some programmer.

A program associates each value with at least one particular type, but it also occurs also that a one value is associated with many subtypes. Other entities, such as objects, modules, communication channels, dependencies can become associated with a type. Even a type can become associated with a type. An implementation of some *type system* could in theory associate some identifications named this way:

- data type – a type of a value
- class – a type of an object
- kind (type theory) – a *type of a type*, or metatype

These are the kinds of abstractions typing can go through on a hierarchy of levels contained in a system.

When a programming language evolves a more elaborate type system, it gains a more finely-grained rule set than basic type checking, but this comes at a price when the type inferences (and other properties) become undecidable, and when more attention must be paid by the programmer to annotate code or to consider computer-related

operations and functioning. It is challenging to find a sufficiently expressive type system that satisfies all programming practices in type safe manner.

The more type restrictions that are imposed by the compiler, the more *strongly typed* a programming language is. Strongly typed languages often require the programmer to make explicit conversions in contexts where an implicit conversion would cause no harm. Pascal's type system has been described as "too strong" because, for example, the size of an array or string is part of its type, making some programming tasks difficult.^{[3][4]} Haskell is also strongly typed but its types are automatically inferred so that explicit conversions are unnecessary.

A programming language compiler can also implement a *dependent type* or an *effect system*, which enables even more program specifications to be verified by a type checker. Beyond simple value-type pairs, a virtual "region" of code is associated with an "effect" component describing *what* is being done *with what*, and enabling for example to "throw" an error report. Thus the symbolic system may be a *type and effect system*, which endows it with more safety checking than type checking alone.

Whether automated by the compiler or specified by a programmer, a type system makes program behavior illegal that is outside the type-system rules. Advantages provided by programmer-specified type systems include:

- *Abstraction* (or *modularity*) – Types enable programmers to think at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can begin to think of a string as a collection of character values instead of as a mere array of bytes. Higher still, types enable programmers to think about and express interfaces between two of *any*-sized subsystems. This enables more levels of localization so that the definitions required for interoperability of the subsystems remain consistent when those two subsystems communicate.
- *Documentation* – In more expressive type systems, types can serve as a form of documentation clarifying the intent of the programmer. For instance, if a programmer declares a function as returning a timestamp type, this documents the function when the timestamp type can be explicitly declared deeper in the code to be integer type.

Advantages provided by compiler-specified type systems include:

- *Optimization* – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of four bytes, the compiler may be able to use more efficient machine instructions.
- *Safety* – A type system enables the compiler to detect meaningless or probably invalid code. For example, we can identify an expression `3 / "Hello, World"` as invalid, when the rules do not specify how to divide an integer by a string. Strong typing offers more safety, but cannot guarantee complete *type safety*.

Type safety contributes to program correctness, but can only guarantee correctness at the expense of making the type checking itself an undecidable problem. In a *type system* with automated type checking a program may prove to run incorrectly yet be safely typed, and produce no compiler errors. Division by zero is an unsafe and incorrect operation, but a type checker running only at compile time doesn't scan for division by zero in most programming languages, and then it is left as a runtime error. To prove the absence of these more-general-than-types defects, other kinds of formal methods, collectively known as program analyses, are in common use. In addition software testing is an empirical method for finding errors that the type checker cannot detect.

Type checking

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

Static typing

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Statically typed languages include ActionScript 3, Ada, C, D, Eiffel, F#, Fortran, Go, Haskell, haXe, JADE, Java, ML, Objective-C, OCaml, Pascal, Seed7 and Scala. C++ is statically typed, aside from its run-time type information system. The C# type system performs static-like compile-time type checking, but also includes full runtime type checking. Perl is statically typed with respect to distinguishing arrays, hashes, scalars, and subroutines.

Static typing is a limited form of program verification (see type safety): accordingly, it allows many type errors to be caught early in the development cycle. Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed. Program execution may also be made more efficient (e.g. faster or taking reduced memory) by omitting runtime type checks and enabling other optimizations.

Because they evaluate type information during compilation and therefore lack type information that is only available at run-time, static type checkers are conservative. They will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed. For example, even if an expression <complex test> always evaluates to true at run-time, a program containing the code

```
if <complex test> then <do something> else <type error>
```

will be rejected as ill-typed, because a static analysis cannot determine that the else branch won't be taken.^[1] The conservative behaviour of static type checkers is advantageous when <complex test> evaluates to false infrequently: A static type checker can detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors, because the combination of all places where values are created and all places where a certain value is used must be taken into account.

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

Dynamic typing

A programming language is said to be dynamically typed when the majority of its type checking is performed at run-time as opposed to at compile-time. In dynamic typing values have types, but variables do not; that is, a variable can refer to a value of any type. Dynamically typed languages include APL, Erlang, Groovy, JavaScript, Lisp, Lua, MATLAB, GNU Octave, Perl (for user-defined types, but not built-in types), PHP, Pick BASIC, Prolog, Python, R, Ruby, Smalltalk and Tcl.

Implementations of dynamically typed languages generally associate run-time objects with "tags" containing their type information. This run-time classification is then used to implement type checks and dispatch overloaded functions, but can also enable pervasive uses of dynamic dispatch, late binding and similar idioms that would be

cumbersome at best in a statically typed language, requiring the use of variant types or similar features.

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

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

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

Development in dynamically typed languages is often supported by programming practices such as unit testing. Testing is a key practice in professional software development, and is particularly important in dynamically typed languages. In practice, the testing done to ensure correct program operation can detect a much wider range of errors than static type-checking, but full test coverage over all possible executions of a program (including timing, user inputs, etc.), if even possible, would be extremely costly and impractical. Static typing helps by providing strong guarantees of a particular subset of commonly made errors never occurring.

Combinations of dynamic and static typing

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

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

As of the 4.0 Release, the .NET Framework supports a variant of dynamic typing via the `System.Dynamic` namespace^[5] whereby a *static* object of type 'dynamic' is a placeholder for the .NET runtime to interrogate its dynamic facilities to resolve the object reference.

Static and dynamic type checking in practice

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

Static typing can find type errors reliably at compile time. This should increase the reliability of the delivered program. However, programmers disagree over how commonly type errors occur, and thus disagree over the proportion of those bugs that are coded that would be caught by appropriately representing the designed types in code. Static typing advocates believe programs are more reliable when they have been well type-checked, while dynamic typing advocates point to distributed code that has proven reliable and to small bug databases. The value of static typing, then, presumably increases as the strength of the type system is increased. Advocates of dependently

typed languages such as Dependent ML and Epigram have suggested that almost all bugs can be considered type errors, if the types used in a program are properly declared by the programmer or correctly inferred by the compiler.^[6]

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

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

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

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

Dynamic typing typically allows duck typing (which enables easier code reuse). Many languages with static typing also feature duck typing or other mechanisms like generic programming which also enables easier code reuse.

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

Strong and weak typing: Liskov Definition

In 1974 Liskov and Zilles described a strong-typed language as one in which "whenever an object is passed from a calling function to a called function, its type must be compatible with the type declared in the called function."^[7] Jackson wrote, "In a strongly typed language each data area will have a distinct type and each process will state its communication requirements in terms of these types."^[8]

Strong and weak typing

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

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

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

In a weakly typed language, the result of this operation depends on language-specific rules. Visual Basic would convert the string "37" into the number 37, perform addition, and produce the number 42. JavaScript would convert the number 5 to the string "5", perform string concatenation, and produce the string "537." In JavaScript, the conversion to string is applied regardless of the order of the operands (for example, `y + x` would be "375") while in AppleScript, the left-most operand determines the type of the result, so that `x + y` is the number 42 but `y + x` is the string "375".

In the same manner, due to JavaScript's dynamic type conversions:

```
var y = 2 / 0;                                // y now equals a constant for
infinity
y == Number.POSITIVE_INFINITY                 // returns true
Infinity == Number.POSITIVE_INFINITY         // returns true
"Infinity" == Infinity                      // returns true
y == "Infinity"                            // returns true
```

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

Safely and unsafely typed systems

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

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

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

In languages like Visual Basic, variable

```
z
```

in the example acquires the value 42. While the programmer may or may not have intended this, the language defines the result specifically, and the program does not crash or assign an ill-defined value to

```
z
```

. In this respect, such languages are type-safe; however, in some languages, if the value of

```
y
```

was a string that could not be converted to a number (e.g. "Hello World"), the results would be undefined. Such languages are type-safe (in that they will not crash), but can easily produce undesirable results. In other languages like JavaScript, the numeric operand would be converted to a string, and then concatenation performed. In this case, the results are not undefined and are predictable.

Now let us look at the same example in C:

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

In this example

z

will point to a memory address five characters beyond

V

, equivalent to three characters after the terminating zero character of the string pointed to by

v

. The content of that location is undefined, and might lie outside addressable memory. The mere computation of such a pointer may result in undefined behavior (including the program crashing) according to C standards, and in typical systems dereferencing z at this point could cause the program to crash. We have a well-typed, but not memory-safe program—a condition that cannot occur in a type-safe language.

In some languages, like JavaScript, the use of special numeric values and constants allows type-safety for mathematical operations without resulting in runtime errors. For example, when dividing a

Number

by a

String

, or a

Number

by zero.

```
var x = 32;  
var aString = new String("A");  
x = x/aString; // x now equals the constant NaN,  
meaning Not A Number  
isNaN(x); // returns true  
typeof(x); // returns "number"  
var y = 2 / 0; // y now equals a constant for infinity  
y == Number.POSITIVE_INFINITY; // returns true  
typeof(y); // returns "number"
```

Variable levels of type checking

Some languages allow different levels of checking to apply to different regions of code. Examples include:-

- The `use strict` directive in Perl applies stronger checking.
- The `@` operator in PHP suppresses some error messages.
- The `Option Strict On` in VB.NET allows the compiler to require a conversion between objects.

Additional tools such as lint and IBM Rational Purify can also be used to achieve a higher level of strictness.

Optional type systems

It has been proposed, chiefly by Gilad Bracha, that the choice of type system be made independent of choice of language; that a type system should be a module that can be "plugged" into a language as required. He believes this is advantageous, because what he calls mandatory type systems make languages less expressive and code more fragile.^[9] The requirement that types do not affect the semantics of the language is difficult to fulfil: for instance, class based inheritance becomes impossible.

Polymorphism and types

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

Duck typing

In "duck typing",^[10] a statement calling a method `m` on an object does not rely on the declared type of the object; only that the object, of whatever type, must supply an implementation of the method called, when called, at run-time.

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

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

Duck typing uses the premise that (referring to a value) "if it walks like a duck, and quacks like a duck, then it is a duck" (this is a reference to the duck test that is attributed to James Whitcomb Riley). The term may have been coined by Alex Martelli in a 2000 message^[11] to the `comp.lang.python` newsgroup (see Python).

Specialized type systems

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

Dependent types

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

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

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

Linear types

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

Intersection types

Intersection types are types describing values that belong to *both* of two other given types with overlapping value sets. For example, in most implementations of C the signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting *either* signed or unsigned chars, because it is compatible with both types.

Intersection types are useful for describing overloaded function types: For example, if "

```
int
```

→

```
int
```

" is the type of functions taking an integer argument and returning an integer, and "

```
float
```

→

```
float
```

" is the type of functions taking a float argument and returning a float, then the intersection of these two types can be used to describe functions that do one or the other, based on what type of input they are given. Such a function could be passed into another function expecting an "

```
int
```

→

```
int
```

" function safely; it simply would not use the "

```
float
```

→

```
float
```

" functionality.

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

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

Union types

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

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

Existential types

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

- $\text{intT} = \{ a: \text{int}; f: (\text{int} \rightarrow \text{int}); \}$
- $\text{floatT} = \{ a: \text{float}; f: (\text{float} \rightarrow \text{int}); \}$

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

In general it's impossible for the typechecker to infer which existential type a given module belongs to. In the above example $\text{intT} \{ a: \text{int}; f: (\text{int} \rightarrow \text{int}); \}$ could also have the type $\exists X \{ a: X; f: (\text{int} \rightarrow \text{int}); \}$. The simplest solution is to

annotate every module with its intended type, e.g.:

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

Although abstract data types and modules had been implemented in programming languages for quite some time, it wasn't until 1988 that John C. Mitchell and Gordon Plotkin established the formal theory under the slogan: "Abstract [data] types have existential type".^[12] The theory is a second-order typed lambda calculus similar to System F, but with existential instead of universal quantification.

Explicit or implicit declaration and inference

Many static type systems, such as those of C and Java, require *type declarations*: The programmer must explicitly associate each variable with a particular type. Others, such as Haskell's, perform *type inference*: The compiler draws conclusions about the types of variables based on how programmers use those variables. For example, given a function

```
f (x, y)
```

that adds

```
x
```

and

```
y
```

together, the compiler can infer that

```
x
```

and

```
y
```

must be numbers – since addition is only defined for numbers. Therefore, any call to

```
f
```

elsewhere in the program that specifies a non-numeric type (such as a string or list) as an argument would signal an error.

Numerical and string constants and expressions in code can and often do imply type in a particular context. For example, an expression

```
3.14
```

might imply a type of floating-point, while

```
[1, 2, 3]
```

might imply a list of integers – typically an array.

Type inference is in general possible, if it is decidable in the type theory in question. Moreover, even if inference is undecidable in general for a given type theory, inference is often possible for a large subset of real-world programs.

Haskell's type system, a version of Hindley-Milner, is a restriction of System F ω to so-called rank-1 polymorphic types, in which type inference is decidable. Most Haskell compilers allow arbitrary-rank polymorphism as an extension, but this makes type inference undecidable. (Type checking is decidable, however, and rank-1 programs still have type inference; higher rank polymorphic programs are rejected unless given explicit type annotations.)

Types of types

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

Types fall into several broad categories:

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

Abstract data types

- Algebraic types
- Subtype
- Derived type
- Object types; e.g., type variable
- Partial type
- Recursive type
- Function types; e.g., binary functions
- universally quantified types, such as parameterized types
- existentially quantified types, such as modules
- Refinement types – types that identify subsets of other types
- Dependent types – types that depend on terms (values)
- Ownership types – types that describe or constrain the structure of object-oriented systems
- Pre-defined types provided for convenience in real-world applications, such as date, time and money.

Unified Type System

Some languages like C# have a unified type system. This means that all C# types including primitive types inherit from a single root object. Every type in C# inherits from the Object class. Java has several primitive types that are not objects. Java provides wrapper object types that exist together with the primitive types so developers can use either the wrapper object types or the simpler non-object primitive types.

Compatibility: equivalence and subtyping

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

If the type of e and the type of x are the same and assignment is allowed for that type, then this is a valid expression. In the simplest type systems, therefore, the question of whether two types are compatible reduces to that of whether they are *equal* (or *equivalent*). Different languages, however, have different criteria for when two type expressions are understood to denote the same type. These different *equational theories* of types vary widely, two extreme cases being *structural type systems*, in which any two types are equivalent that describe values with the same structure, and *nominative type systems*, in which no two syntactically distinct type expressions denote the same type (*i.e.*, types must have the same "name" in order to be equal).

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

Programming style

Some programmers prefer statically typed languages; others prefer dynamically typed languages. Statically typed languages alert programmers to type errors during compilation, and they may perform better at runtime. Advocates of dynamically typed languages claim they better support rapid prototyping and that type errors are only a small subset of errors in a program.^{[13][14]} Likewise, there is often no need to manually declare all types in statically typed languages with type inference; thus, the need for the programmer to explicitly specify types of variables is automatically lowered for such languages; and some dynamic languages have run-time optimisers^{[15][16]} that can generate fast code approaching the speed of static language compilers, often by using partial type inference.

References

- [1] Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press. ISBN 0-262-16209-1.
- [2] Pierce, Benjamin C. (2002), p. 208
- [3] Infoworld 25 April 1983 (http://books.google.co.uk/books?id=7i8EAAAAMBAJ&pg=PA66&lpg=PA66&dq=pascal+type+system+"too+strong"&source=bl&ots=PGyKS1fWUb&sig=ebFl6fk_yxwyY4b7sHSklp048Q4&hl=en&ei=lSmjTunuBo6F8gPOu43CCA&sa=X&oi=book_result&ct=result&resnum=1&ved=0CBsQ6AEwAA#v=onepage&q=pascal type system "too strong"&f=false)
- [4] [[Brian Kernighan (<http://www.cs.virginia.edu/~cs655/readings/bwk-on-pascal.html>): *Why Pascal is not my favorite language*]
- [5] [http://msdn.microsoft.com/en-us/library/dd233052\(VS.100\).aspx](http://msdn.microsoft.com/en-us/library/dd233052(VS.100).aspx)
- [6] Xi, Hongwei; Scott, Dana (1998). "Dependent Types in Practical Programming". *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages* (ACM Press): 214–227. CiteSeerX: 10.1.1.41.548 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.41.548>).
- [7] Liskov, B; Zilles, S (1974). "Programming with abstract data types". *ACM Sigplan Notices*. CiteSeerX: 10.1.1.136.3043 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.136.3043>).
- [8] Jackson, K. (1977). "Parallel processing and modular software construction" (<http://www.springerlink.com/content/wq02703237400667/>). *Lecture Notes in Computer Science* 54: 436–443. doi:10.1007/BFb0021435..
- [9] Bracha, G.: *Pluggable Types* (<http://bracha.org/pluggableTypesPosition.pdf>)
- [10] Rozsnyai, S.; Schiefer, J.; Schatten, A. (2007). "Concepts and models for typing events for event-based systems". *Proceedings of the 2007 inaugural international conference on Distributed event-based systems - DEBS '07*. pp. 62. doi:10.1145/1266894.1266904. ISBN 9781595936653.
- [11] Martelli, Alex (26 July 2000). "lmvn60171@news1.newsguy.com Re: polymorphism (was Re: Type checking in python?) (news:8)". Web link (<http://groups.google.com/group/comp.lang.python/msg/e230ca916be58835?hl=en&>).
- [12] Mitchell, John C.; Plotkin, Gordon D.; *Abstract Types Have Existential Type* (<http://theory.stanford.edu/~jcm/papers/mitch-plotkin-88.pdf>), ACM Transactions on Programming Languages and Systems, Vol. 10, No. 3, July 1988, pp. 470–502
- [13] Meijer, Erik; Drayton, Peter. "Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages" (<http://research.microsoft.com/en-us/um/people/emeijer/Papers/RDL04Meijer.pdf>). Microsoft Corporation. .
- [14] Eckel, Bruce. "Strong Typing vs. Strong Testing" (http://docs.google.com/View?id=dcsvntt2_25wpjvbbhk). Google Docs. .
- [15] "Adobe and Mozilla Foundation to Open Source Flash Player Scripting Engine" (<http://www.mozilla.com/en-US/press/mozilla-2006-11-07.html>). .
- [16] "Psyco, a Python specializing compiler" (<http://psyco.sourceforge.net/introduction.html>). .

Further reading

- Cardelli, Luca (1997). "Type systems". In Allen B. Tucker. *CRC Handbook of Computer Science and Engineering*. CRC Press.
- Smith, Chris, *What To Know Before Debating Type Systems* (<http://cdsmith.wordpress.com/2011/01/09/an-old-article-i-wrote/>)
- Tratt, Laurence, *Dynamically Typed Languages* (http://tratt.net/laurie/research/publications/html/tratt_dynamically_typed_languages/), Advances in Computers, Vol. 77, pp. 149–184, July 2009

Scope (computer science)

In computer programming, a **scope** is the context within a computer program in which a variable name or other identifier is valid and can be used, or within which a declaration has effect. Outside of the scope of a variable name, the variable's value may still be stored, and may even be accessible in some way, but the name does not refer to it; that is, the name is not *bound* to the variable's storage.

Various programming languages have various different scoping rules for different kinds of declarations and identifiers. Such scoping rules have a large effect on language semantics and, consequently, on the behavior and correctness of programs. In languages like C++, accessing an unbound variable does not have well-defined semantics and may result in undefined behavior; and declarations or identifiers used outside their scope will generate syntax errors.

Scopes are frequently tied to other language constructs, but many languages also offer constructs specifically for controlling scope.

Scope within a function

Function scope

```
function square(n) {
    return n * n;
}

function sum_of_squares(n) {
    var ret = 0; // the value to return
    var i = 1; // a counter to go from 1 to n
    while(i <= n) {
        ret = ret + square(i);
        i = i + 1;
    }
    return ret;
}
```

Most of the commonly used programming languages offer a way to create a *local variable* in a function: a variable that, in some sense, disappears when the function returns.

For example, in the snippet of JavaScript code on the right, two functions are defined: `square` and `sum_of_squares`. `square` computes the square of a number; `sum_of_squares` computes the sum of all squares up to a number. (For example, `square(4)` is $4^2 = 16$, and `sum_of_squares(4)` is $1^2 + 2^2 + 3^2 + 4^2 = 30$.)

Each of these functions has a variable named `n` that represents the argument to the function. These two `n` variables are completely separate and unrelated, despite having the same name, because they are local variables, with *function scope*: each one's scope is its own function, so they don't overlap. Therefore, `sum_of_squares` can call `square` without its own `n` being altered. Similarly, `sum_of_squares` has variables named `ret` and `i`; these variables, because of their limited scope, will not interfere with any variables named `ret` or `i` that might belong to any other function. In other words, there is no risk of a *name collision* between these identifiers and any unrelated identifiers, even if they are identical.

Block scope within a function

```
int sum_of_squares(int n) {
    int ret = 0;
    int i = 1;
    while(i <= n) {
        int n = i * i;
        ret = ret + n;
        i = i + 1;
    }
    return ret;
}
```

Many languages take function scope slightly further, allowing variables to be made local to just *part* of a function; rather than having the entire function as its scope, a variable might have *block scope*, meaning that it's scoped to just a single block of statements. This is demonstrated in the C code at right, which has two distinct local variables named `n`: one whose scope is the entire function, and one that exists only inside the `while`-loop. In fact, each iteration of the `while`-loop creates a new instance of `n`. This inner `n` completely "hides" or "shadows" the outer `n`, such that the outer `n` is invisible inside the `while`-loop — but can still be used in the loop header, `while(i <= n)`, which always refers to the outer `n`.

This is a rather contrived example; real-world C programmers do not usually name two local variables in such a way that one hides the other. Furthermore, some descendants of C, such as Java and C#, despite having support for block scope (in that a local variable can be made to go out of scope before the end of a function), do not allow one local variable to hide another. In such languages, the attempted declaration of the second `n` would result in a syntax error, and one of the `n` variables would have to be renamed.

Since the main use of blocks within a function is in control structures such as `if`-statements and `while`-loops, block scope tends to link the scope of variables to the structure of a function's flow of execution. However, languages with block scope typically also allow the use of "naked" blocks, which frequently serve no other purpose than to allow fine-grained control of variable scope.

Let-expressions

Many languages, especially functional languages, offer a feature called *let-expressions*, which allow a declaration's scope to be a single expression. This is convenient if, for example, an intermediate value is needed for a computation. For example, in Standard ML, if `f()` returns 12, then `let val x = f() in x * x end` is an expression that evaluates to 144, using a temporary variable named `x` to avoid calling `f()` twice. Some languages with block scope approximate this functionality by offering syntax for a block to be embedded into an expression; for example, the aforementioned Standard ML expression could be written in Perl as `do { my $x = f(); $x * $x } , or in GNU C as ({ int x = f(); x * x; }) .`

Scope outside a function

A declaration has *global scope* if it has effect throughout an entire program, or (in some languages) if it has effect from the point of its occurrence until the end of the source-file it occurs in. The latter is also called *file scope*. Variable names with global scope — called *global variables* — are frequently considered bad practice, at least in some languages; but global scope is typically used (depending on the language) for various other sorts of identifiers, such as names of functions, and names of classes and other data types. In the JavaScript snippet we saw above, the function-names `square` and `sum_of_squares` have truly global scope, while in the C snippet, the function-name `sum_of_squares` has file scope.

```
{ my $counter = 0;
  sub increment_counter()
  { $counter = $counter + 1;
    return $counter;
  }
}
```

Some languages allow the concept of block scope to be applied, to varying extents, outside of a function. For example, in the Perl snippet at right, `$counter` is a variable name with block scope (due to the use of the `my` keyword), while `increment_counter` is a function name with global scope. Each call to `increment_counter` will increase the value of `$counter` by one, and return the new value. Code outside of this block can call `increment_counter`, but cannot otherwise obtain or alter the value of `$counter`.

As we saw above, function scope and block scope are very useful for avoiding name collisions, but even at global scope, many languages have mechanisms such as namespaces to help mitigate this problem.

Lexical scoping and dynamic scoping

As we have seen, the use of local variables — of variable names with limited scope, that only exist within a specific function — helps avoid the risk of a name collision between two identically named variables. However, we have sidestepped an important question: what does it mean to be "within" a function? There are two very different approaches to scoping that answer this question in different ways.

In *lexical scoping* (or *lexical scope*; also called *static scoping* or *static scope*), if a variable name's scope is a certain function, then its scope is the program text of the function definition: within that text, the variable name exists, and is bound to its variable, but outside that text, the variable name does not exist. By contrast, in *dynamic scoping* (or *dynamic scope*), if a variable name's scope is a certain function, then its scope is the time-period during which the function is executing: while the function is running, the variable name exists, and is bound to its variable, but after the function returns, the variable name does not exist. This means that if function `f` invokes a separately defined function `g`, then under lexical scoping, function `g` does *not* have access to `f`'s local variables (since the text of `g` is not inside the text of `f`), while under dynamic scoping, function `g` *does* have access to `f`'s local variables (since the invocation of `g` is inside the invocation of `f`).

```
x=1
function g () { echo $x ; x=2 ; }
function f () { local x=3 ; g ; }
f # does this print 1, or 3?
echo $x # does this print 1, or 2?
```

Consider, for example, the program at right. The first line, `x=1`, creates a global variable `x` and initializes it to 1. The second line, `function g () { echo $x ; x=2 ; }`, defines a function `g` that prints out ("echoes") the current value of `x`, and then sets `x` to 2 (overwriting the previous value). The third line, `function f () {`

`local x=3 ; g ; }` defines a function `f` that creates a local variable `x` (hiding the identically named global variable) and initializes it to 3, and then calls `g`. The fourth line, `f`, calls `f`. The fifth line, `echo $x`, prints out the current value of `x`.

So, what exactly does this program print? It depends on the scoping rules. If the language of this program is one that uses lexical scoping, then `g` prints and modifies the global variable `x` (because `g` is defined outside `f`), so the program prints 1 and then 2. By contrast, if this language uses dynamic scoping, then `g` prints and modifies `f`'s local variable `x` (because `g` is called from within `f`), so the program prints 3 and then 1. (As it happens, the language of the program is Bash, which uses dynamic scoping; so the program prints 3 and then 1.)

Lexical scoping

With **lexical scope**, a name always refers to its (more or less) local lexical environment. This is a property of the program text and is made independent of the runtime call stack by the language implementation. Because this matching only requires analysis of the static program text, this type of scoping is also called **static scoping**. Lexical scoping is standard in all ALGOL-based languages such as Pascal, Modula2 and Ada as well as in modern functional languages such as ML and Haskell. It is also used in the C language and its syntactic and semantic relatives, although with different kinds of limitations. Static scoping allows the programmer to reason about object references such as parameters, variables, constants, types, functions, etc. as simple name substitutions. This makes it much easier to make modular code and reason about it, since the local naming structure can be understood in isolation. In contrast, dynamic scope forces the programmer to anticipate all possible dynamic contexts in which the module's code may be invoked.

```
program A;
var I:integer;
K:char;

procedure B;
var K:real;
L:integer;

procedure C;
var M:real;
begin
(*scope A+B+C*)
end;

(*scope A+B*)
end;

(*scope A*)
end.
```

For example, consider the Pascal program fragment at right. The variable `I` is visible at all points, because it is never hidden by another variable of the same name. The `char` variable `K` is visible only in the main program because it is hidden by the `real` variable `K` visible in procedure `B` and `C` only. Variable `L` is also visible only in procedure `B` and `C` but it does not hide any other variable. Variable `M` is only visible in procedure `C` and therefore not accessible either from procedure `B` or the main program. Also, procedure `C` is visible only in procedure `B` and can therefore not be called from the main program.

There could have been another procedure `C` declared in the program outside of procedure `B`. The place in the program where "`C`" is mentioned then determines which of the two procedures named `C` it represents, thus precisely analogous with the scope of variables.

Correct implementation of static scope in languages with first-class nested functions is not trivial, as it requires each function value to carry with it a record of the values of the variables that it depends on (the pair of the function and this environment is called a closure). Depending on implementation and computer architecture, variable lookup *may* become slightly inefficient when very deeply lexically nested functions are used, although there are well-known techniques to mitigate this. Also, for nested functions that only refer to their own arguments and (immediately) local variables, all relative locations can be known at compile time. No overhead at all is therefore incurred when using that type of nested function. The same applies to particular parts of a program where nested functions are not used, and, naturally, to programs written in a language where nested functions are not available (such as in the C language).

History

Lexical scoping was used for ALGOL and has been picked up in most other languages since then. *Deep binding*, which approximates static (lexical) scoping, was introduced in LISP 1.5 (via the Funarg device developed by Steve Russell, working under John McCarthy). The original Lisp interpreter (1960) and most early Lisps used dynamic scoping, but descendants of dynamically scoped languages often adopt static scoping; Common Lisp has both dynamic and static scoping while Scheme uses static scoping exclusively. Perl is another language with dynamic scoping that added static scoping afterwards. Languages like Pascal and C have always had lexical scoping, since they are both influenced by the ideas that went into ALGOL 60 (although C did not include lexically nested functions).

Dynamic scoping

With **dynamic scope**, each identifier has a global stack of bindings. Introducing a local variable with name `x` pushes a binding onto the global `x` stack (which may have been empty), which is popped off when the control flow leaves the scope. Evaluating `x` in any context always yields the top binding. In other words, a global identifier refers to the identifier associated with the most recent environment. Note that this cannot be done at compile-time because the binding stack only exists at run-time, which is why this type of scoping is called *dynamic* scoping.

Generally, certain blocks are defined to create bindings whose lifetime is the execution time of the block; this adds some features of static scoping to the dynamic scoping process. However, since a section of code can be called from many different locations and situations, it can be difficult to determine at the outset what bindings will apply when a variable is used (or if one exists at all). This can be beneficial; application of the principle of least knowledge suggests that code avoid depending on the *reasons* for (or circumstances of) a variable's value, but simply use the value according to the variable's definition. This narrow interpretation of shared data can provide a very flexible system for adapting the behavior of a function to the current state (or policy) of the system. However, this benefit relies on careful documentation of all variables used this way as well as on careful avoidance of assumptions about a variable's behavior, and does not provide any mechanism to detect interference between different parts of a program. Dynamic scoping also voids all the benefits of referential transparency. As such, dynamic scoping can be dangerous and few modern languages use it. Some languages, like Perl and Common Lisp, allow the programmer to choose static or dynamic scoping when defining or redefining a variable. Logo and Emacs lisp are examples of languages that use dynamic scoping.

Dynamic scoping is fairly easy to implement. To find an identifier's value, the program could traverse the runtime stack, checking each activation record (each function's stack frame) for a value for the identifier. In practice, this is made more efficient via the use of an association list, which is a stack of name/value pairs. Pairs are pushed onto this stack whenever declarations are made, and popped whenever variables go out of scope.^[1] *Shallow binding* is an

alternative strategy that is considerably faster, making use of a *central reference table*, which associates each name with its own stack of meanings. This avoids a linear search during run-time to find a particular name, but care should be taken to properly maintain this table.^[2] Note that both of these strategies assume a last-in-first-out (LIFO) ordering to bindings for any one variable; in practice all bindings are so ordered.

An even simpler implementation is the representation of dynamic variables with simple global variables. The local binding is performed by saving the original value in an anonymous location on the stack that is invisible to the program. When that binding scope terminates, the original value is restored from this location. In fact, dynamic scope originated in this manner. Early implementations of Lisp used this obvious strategy for implementing local variables, and the practice survives in some dialects which are still in use, such as GNU Emacs Lisp. Lexical scope was introduced into Lisp later. This is equivalent to the above shallow binding scheme, except that the central reference table is simply the global variable binding environment, in which the current meaning of the variable is its global value. Maintaining global variables isn't complex. For instance, a symbol object can have a dedicated slot for its global value.

Dynamic scoping provides an excellent abstraction for thread local storage, but if it is used that way it cannot be based on saving and restoring a global variable. A possible implementation strategy is for each variable to have a thread-local key. When the variable is accessed, the thread-local key is used to access the thread-local memory location (by code generated by the compiler, which knows which variables are dynamic and which are lexical). If the thread-local key does not exist for the calling thread, then the global location is used. When a variable is locally bound, the prior value is stored in a hidden location on the stack. The thread-local storage is created under the variable's key, and the new value is stored there. Further nested overrides of the variable within that thread simply save and restore this thread-local location. When the initial, outer-most override's scope terminates, the thread-local key is deleted, exposing the global version of the variable once again to that thread.

Qualified identifiers

As we have seen, one of the key reasons for scope is that it helps prevent name collisions, by allowing identical identifiers to refer to distinct things, with the restriction that the identifiers must have separate scopes. Sometimes this restriction is inconvenient; when many different things need to be accessible throughout a program, they generally all need identifiers with global scope, so different techniques are required to avoid name collisions.

To address this, many languages offer mechanisms for organizing global identifiers. The details of these mechanisms, and the terms used, depend on the language; but the general idea is that a group of identifiers can itself be given a name — a prefix — and, when necessary, an entity can be referred to by a *qualified identifier* consisting of the identifier plus the prefix. Normally such identifiers will have, in a sense, two sets of scopes: a scope (usually the global scope) in which the qualified identifier is visible, and one or more narrower scopes in which the *unqualified identifier* (without the prefix) is visible as well. And normally these groups can themselves be organized into groups; that is, they can be *nested*.

Although many languages support this concept, the details vary greatly. Some languages have mechanisms, such as *namespaces* in C++ and C#, that serve almost exclusively to enable global identifiers to be organized into groups. Other languages have mechanisms, such as *packages* in Ada and *structures* in Standard ML, that combine this with the additional purpose of allowing some identifiers to be visible only to other members of their group. And object-oriented languages often allow classes or singleton objects to fulfill this purpose (whether or not they *also* have a mechanism for which this is the primary purpose). Furthermore, languages often meld these approaches; for example, Perl's packages are largely similar to C++'s namespaces, but optionally double as classes for object-oriented programming; and Java organizes its variables and functions into classes, but then organizes those classes into Ada-like packages.

References

- [1] Scott 2006, p. 135
- [2] Scott 2006, p. 135

Further reading

- Harold Abelson and Gerald Jay Sussman. "Lexical addressing" (http://mitpress.mit.edu/sicp/full-text/book/book-Z-H-35.html#%_sec_5.5.6). *Structure and Interpretation of Computer Programs*.
- Scott, M. (2006). *Programming Language Pragmatics*, 2nd edition. Morgan Kauffman Publishers, San Francisco, CA.

Masterpiece of Alan Kay

Class (computer programming)

In object-oriented programming, a **class** is a construct that is used to create instances of itself – referred to as *class instances*, *class objects*, *instance objects* or simply *objects*. A class defines constituent members which enable its instances to have state and behavior.^[1] Data field members (*member variables* or *instance variables*) enable a class instance to maintain state. Other kinds of members, especially *methods*, enable the behavior of class instances. Classes define the type of their instances.^[2]

A class usually represents a noun, such as a person, place or thing, or something nominalized. For example, a "Banana" class would represent the properties and functionality of bananas in general. A single, particular banana would be an instance of the "Banana" class, an object of the type "Banana".

Design and implementation

Classes are composed from structural and behavioral constituents.^[3] Programming languages that include classes as a programming construct offer support for various class-related features, and the syntax required to use these features vary greatly from one programming language to another.

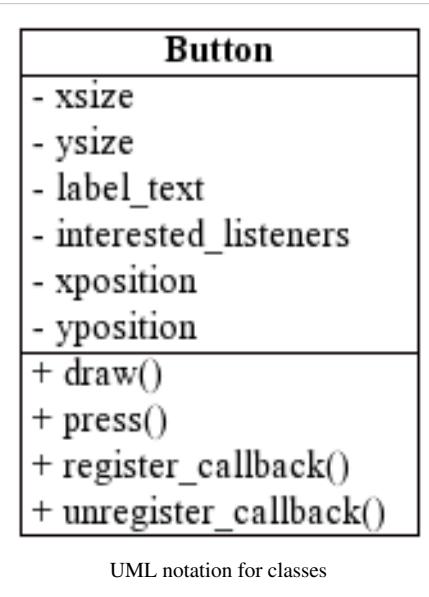
Structure

A class contains data field descriptions (or *properties*, *fields*, *data members*, or *attributes*). These are usually field types and names that will be associated with state variables at program run time; these state variables either belong to the class or specific instances of the class. In most languages, the structure defined by the class determines the layout of the memory used by its instances. Other implementations are possible, for example, objects in Python use associative key-value containers.^[4]

Some programming languages support specification of invariants as part of the definition of the class, and enforce them through the type system. Encapsulation of state is necessary for being able to enforce the invariants of the class.

Behavior

The behavior of class or its instances is defined using methods. Methods are subroutines with the ability to operate on objects or classes. These operations may alter the state of an object or simply provide ways of accessing it.^[5] Many kinds of methods exist, but support for them varies across languages. Some types of methods are created and called by programmer code, while other special methods—such as constructors, destructors, and conversion operators—are created and called by compiler-generated code. A language may also allow the programmer to define and call these special methods.^{[6][7]}



The concept of *class interface*

Every class **implements** (or *realizes*) an interface by providing structure and behavior. Structure consists of data and state, and behavior consists of code that specifies how methods are implemented.^[8] There is a distinction between the definition of an interface and the implementation of that interface; however, this line is blurred in many programming languages because class declarations both define and implement an interface. Some languages, however, provide features that separate interface and implementation. For example, an abstract class can define an interface without providing implementation.

Languages that support class inheritance also allow classes to inherit interfaces from the classes that they are derived from. In languages that support access specifiers, the interface of a class is considered to be the set of public members of the class, including both methods and attributes (via implicit getter and setter methods); any private members or internal data structures are not intended to be depended on by external code and thus are not part of the interface.

Object-oriented programming methodology dictates that the operations of any interface of a class are to be independent of each other. It results in a layered design where clients of an interface use the methods declared in the interface. An interface places no requirements for clients to invoke the operations of one interface in any particular order. This approach has the benefit that client code can assume that the operations of an interface are available for use whenever the client has access to the object.

Example

The buttons on the front of your television set are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to toggle the television on and off. In this example, your particular television is the instance, each method is represented by a button, and all the buttons together comprise the interface. (Other television sets that are the same model as yours would have the same interface.) In its most common form, an interface is a specification of a group of related methods without any associated implementation of the methods.

A television set also has a myriad of *attributes*, such as size and whether it supports color, which together comprise its structure. A class represents the full description of a television, including its attributes (structure) and buttons (interface).

Getting the total number of televisions manufactured could be a *static method* of the television class. This method is clearly associated with the class, yet is outside the domain of each individual instance of the class. Another example would be a static method that finds a particular instance out of the set of all television objects.

Member accessibility

Further information: Information hiding

Many languages support the concept of **information hiding** and **encapsulation**, typically with **access specifiers** for class members. Access specifiers control access to class members. Some access specifiers may also control how classes inherit such constraints. Their primary purpose is to separate the interface of a class from its implementation.

The following is a common set of access specifiers:^[9]

- **private** (or *class-private*) restricts the access to the class itself. Only methods that are part of the same class can access private members.
- **protected** (or *class-protected*) allows the class itself and all its subclasses to access the member.
- **public** means that any code can access the member by its name.

Although many object-oriented languages support the above access specifiers, their semantics may differ.

Object-oriented design uses the access specifiers in conjunction with careful design of public method implementations to enforce class invariants—constraints on the state of the objects. A common usage of access

specifiers is to separate the internal data of a class from its interface: the internal structure is made private, while public accessor methods can be used to inspect or alter such private data.

Access specifiers do not necessarily control **visibility**, in that even private members may be visible to client external code. In some languages, an inaccessible but visible member may be referred to at run-time (for example, by a pointer returned from a member function), but an attempt to use it by referring to the name of the member from client code will be prevented by the type checker.^[10]

The various object-oriented programming languages enforce member accessibility and visibility to various degrees, and depending on the language's type system and compilation policies, enforced at either compile-time or run-time. For example, the Java language does not allow client code that accesses the private data of a class to compile.^[11] In the C++ language, private methods are visible, but not accessible in the interface; however, they may be made invisible by explicitly declaring fully abstract classes that represent the interfaces of the class.^[12]

Some languages feature other accessibility schemes:

- **Instance vs. class accessibility:** Ruby supports **instance-private** and **instance-protected** access specifiers in lieu of class-private and class-protected, respectively. They differ in that they restrict access based on the instance itself, rather than the instance's class.^[13]
- **Friend:** C++ supports a mechanism where a function explicitly declared as a friend function of the class may access the members designated as private or protected.^[14]
- **Path-based:** Java supports restricting access to a member within a Java package, which is roughly the path of a file.^[9]

Inter-class relationships

In addition to the design of standalone classes, programming languages may support more advanced class design based upon relationships between classes. The inter-class relationship design capabilities commonly provided are *compositional* and *hierarchical*.

Compositional

Classes can be composed of other classes, thereby establishing a compositional relationship between the enclosing class and its embedded classes. Compositional relationship between classes is also commonly known as a **has-a** relationship.^[15] For example, a class "Car" could be composed of and contain a class "Engine". Therefore, a Car **has an** Engine. One aspect of composition is containment, which is the enclosure of component instances by the instance that has them. If an enclosing object contains component instances by value, the components and their enclosing object have a similar lifetime. If the components are contained by reference, they may not have a similar lifetime.^[16]

Example (Objective-C 2.0 code):

```
@class Engine;

@interface Car : NSObject

@property (nonatomic, retain) NSString *name;
@property (nonatomic, retain) Engine *engine
@property (nonatomic, retain) NSArray *tyres;

@end
```

This Car class **have** an instance of NSString, an Engine, and an NSArray.

Hierarchical

Classes can be *derived* from one or more existing classes, thereby establishing a hierarchical relationship between the derived-from classes (*base classes*, *parent classes* or *superclasses*) and the derived class (*child class* or *subclass*). The relationship of the derived class to the derived-from classes is commonly known as an **is-a** relationship.^[17] For example, a class 'Button' could be derived from a class 'Control'. Therefore, a Button **is a** Control. Structural and behavioral members of the parent classes are *inherited* by the child class. Derived classes can define additional structural members (data fields) and/or behavioral members (methods) in addition to those that they *inherit* and are therefore *specializations* of their superclasses. Also, derived classes can override inherited methods if the language allows.

Not all languages support multiple inheritance. For example, Java allows a class to implement multiple interfaces, but only inherit from one class.^[18] If multiple inheritance is allowed, the hierarchy is a directed acyclic graph (or DAG for short), otherwise it is a tree. The hierarchy has classes as nodes and inheritance relationships as links. Classes in the same level are more likely to be associated than classes in different levels. The levels of this hierarchy are called layers or levels of abstraction.

Example (Simplified Objective-C 2.0 code, from iPhone SDK):

```
@interface UIResponder : NSObject //...
@interface UIView : UIResponder //...
@interface UIScrollView : UIView //...
@interface UITableView : UIScrollView //...
```

In this example, a UITableView **is a** UIScrollView **is a** UIView **is a** UIResponder **is an** NSObject.

Definitions of *subclass*

Conceptually, a superclass should be considered as a common part of its subclasses. This factoring of commonality is one mechanism for providing reuse. Thus, extending a superclass by modifying the existing class is also likely to narrow its applicability in various situations. In object-oriented design, careful balance between applicability and functionality of superclasses should be considered. Subclassing is different from subtyping in that subtyping deals with common behavior whereas subclassing is concerned with common structure.

There are two different points of view as to whether subclasses of the same class are required to be disjoint. Sometimes, subclasses of a particular class are considered to be completely disjoint. That is, every instance of a class has exactly one *most-derived class*, which is a subclass of every class that the instance has. This view does not allow dynamic change of object's class, as objects are assumed to be created with a fixed most-derived class. The basis for not allowing changes to object's class is that the class is a compile-time type, which does not usually change at runtime, and polymorphism is utilized for any dynamic change to the object's behavior, so this ability is not necessary. Design that does not need to perform changes to object's type will be more robust and easy-to-use from the point of view of the users of the class.

From another point of view, subclasses are not required to be disjoint. Then there is no concept of a most-derived class, and all types in the inheritance hierarchy that are types of the instance are considered to be equally types of the instance. This view is based on a dynamic classification of objects, such that an object may change its class at runtime. Then object's class is considered to be its *current* structure, but changes to it are allowed. The basis for allowing changes to object's class is a perceived inconvenience caused by replacing an instance with another instance of a different type, since this would require change of all references to the original instance to be changed to refer to the new instance. When changing the object's class, references to the existing instances do not need to be replaced with references to new instances when the class of the object changes. However, this ability is not readily available in all programming languages. This analysis depends on the proposition that dynamic changes to object structure are common. This may or may not be the case in practice.

Orthogonality of the class concept and inheritance

Although class-based languages are commonly assumed to support inheritance, inheritance is not an intrinsic aspect of the concept of classes. Some languages, often referred to as "object-based languages", support classes yet do not support inheritance. Examples of object-based languages include earlier versions of Visual Basic.

Within object-oriented analysis

In object-oriented analysis and in UML, an **association** between two classes represents a collaboration between the classes or their corresponding instances. Associations have direction; for example, a bi-directional association between two classes indicates that both of the classes are aware of their relationship.^[19] Associations may be labeled according to their name or purpose.^[20]

An association role is given end of an association and describes the role of the corresponding class. For example, a "subscriber" role describes the way instances of the class "Person" participate in a "subscribes-to" association with the class "Magazine". Also, a "Magazine" has the "subscribed magazine" role in the same association. Association role multiplicity describes how many instances correspond to each instance of the other class of the association. Common multiplicities are "0..1", "1..1", "1..*" and "0..*", where the "*" specifies any number of instances.^[19]

Taxonomy of classes

There are many categories of classes; however, these categories do not necessarily divide classes into distinct partitions.

Abstract and Concrete

In a language that supports inheritance, an **abstract class**, or *abstract base class* (ABC), is a class that cannot be instantiated because it is either labeled as abstract or it simply specifies abstract methods (or *virtual methods*). Abstract classes specify virtual methods via signatures that are to be implemented by direct or indirect descendants of the abstract class. Before a class derived from an abstract class can be instantiated, all abstract methods of its parent classes must be implemented by some class in the derivation chain.^[21]

Most object oriented programming languages allow the programmer to specify which classes are considered abstract and will not allow these to be instantiated. For example, in Java and PHP, the keyword *abstract* is used.^{[22][23]} In C++, an abstract class is a class having at least one abstract method given by the appropriate syntax in that language (a pure virtual function in C++ parlance).^[21]

A class consisting of only virtual methods is called a Pure Abstract Base Class (or *Pure ABC*) in C++ and is also known as an *interface* by users of the language.^[12] Other languages, notably Java and C#, support a variant of abstract classes called an interface via a keyword in the language. In these languages, multiple inheritance is not allowed, but a class can implement multiple interfaces. Such a class can only contain abstract publicly accessible methods.^{[18] [24] [25]}

A **concrete class** is a class that can be instantiated, as opposed to abstract classes, which cannot.

Local and inner

In some languages, classes can be declared in scopes other than the global scope. There are various types of such classes.

An **Inner class** is a class defined within another class. The relationship between an inner class and its containing class can also be treated as another type of class association. An inner class is typically neither associated with instances of the enclosing class nor instantiated along with its enclosing class. Depending on language, it may or may not be possible to refer to the class from outside the enclosing class. A related concept is *inner types*, also known as *inner data type* or *nested type*, which is a generalization of the concept of inner classes. C++ is an example

of a language that supports both inner classes and inner types (via *typedef* declarations). [26] [27]

Another type is a **local class**, which is a class defined within a procedure or function. This limits references to the class name to within the scope where the class is declared. Depending on the semantic rules of the language, there may be additional restrictions on local classes compared non-local ones. One common restriction is to disallow local class methods to access local variables of the enclosing function. For example, in C++, a local class may refer to static variables declared within its enclosing function, but may not access the function's automatic variables. [28]

Metaclasses

Metaclasses are classes whose instances are classes. [29] A metaclass describes a common structure of a collection of classes and can implement a design pattern or describe particular kinds of classes. Metaclasses are often used to describe frameworks.

In some languages, such as Python, Ruby or Smalltalk, a class is also an object; thus each class is an instance of a unique metaclass which is built into the language. [4] [30] [31] For example, in Objective-C, each object and class is an instance of NSObject. [32] The Common Lisp Object System (CLOS) provides metaobject protocols (MOPs) to implement those classes and metaclasses. [33]

Non-subclassable

Non-subclassable classes allow programmers to design classes and hierarchies of classes which at some level in the hierarchy, further derivation is prohibited. (A stand-alone class may be also designated as non-subclassable, preventing the formation of any hierarchy). Contrast this to *abstract* classes, which imply, encourage, and require derivation in order to be used at all. A non-subclassable class is implicitly *concrete*.

A non-subclassable class is created by declaring the class as `sealed` in C# or as `final` in Java. [34] [35] For example, Java's

```
String
```

class is designated as *final*. [36]

Non-subclassable classes may allow a compiler (in compiled languages) to perform optimizations that are not available for subclassable classes.

Partial

In languages supporting the feature, a **partial class** is a class whose definition may be split into multiple pieces, within a single source-code file or across multiple files. The pieces are merged at compile-time, making compiler output the same as for a non-partial class.

The primary motivation for introduction of partial classes is to facilitate the implementation of code generators, such as visual designers. It is otherwise a challenge or compromise to develop code generators that can manage the generated code when it is interleaved within developer-written code. Using partial classes, a code generator can process a separate file or coarse-grained partial class within a file, and is thus alleviated from intricately interjecting generated code via extensive parsing, increasing compiler efficiency and eliminating the potential risk of corrupting developer code. In a simple implementation of partial classes, the compiler can perform a phase of precompilation where it "unifies" all the parts of a partial class. Then, compilation can proceed as usual.

Other benefits and effects of the partial class feature include:

- Enables separation of a class's interface and implementation code in a unique way.
- Eases navigation through large classes within a editor.

- Enables separation of concerns, in a way similar to aspect-oriented programming but without using any extra tools.
- Enables multiple developers to work on a single class concurrently without the need to merge individual code into one file at a later time.

Partial classes have existed in Smalltalk under the name of *Class Extensions* for considerable time. With the arrival of the .NET framework 2, Microsoft introduced partial classes, supported in both C# 2.0 and Visual Basic 2005. WinRT also supports partial classes.

Example in VB.NET

This simple example, written in Visual Basic .NET, shows how parts of the same class are defined in two different files.

file1.vb

```
Partial Class MyClass
    Private _name As String
End Class
```

file2.vb

```
Partial Class MyClass
    Public Readonly Property Name() As String
        Get
            Return _name
        End Get
    End Property
End Class
```

When compiled, the result is the same as if the two files were written as one, like this:

```
Class MyClass
    Private _name As String
    Public Readonly Property Name() As String
        Get
            Return _name
        End Get
    End Property
End Class
```

Example in Objective-C

In Objective-C, partial classes, aka **categories** may even spread over multiple libraries and executables, like this example:

In Foundation, header file NSData.h:

```
@interface NSData : NSObject

- (id)initWithContentsOfURL:(NSURL *)URL;
//...

@end
```

In user-supplied library, a separate binary from Foundation framework, header file NSData+base64.h:

```
#import <Foundation/Foundation.h>

@interface NSData (base64)

- (NSString *)base64String;
- (id)initWithBase64String:(NSString *)base64String;

@end
```

And in an app, yet another separate binary file, source code file main.m:

```
#import <Foundation/Foundation.h>
#import "NSData+base64.h"

int main(int argc, char *argv[])
{
    if (argc < 2)
        return EXIT_FAILURE;
    NSString *sourceURLString = [NSString stringWithCString:argv[1]];
    NSData *data = [[NSData alloc] initWithContentsOfURL:[NSURL
URLWithString:sourceURLString]];
    NSLog(@"%@", [data base64String]);
    return EXIT_SUCCESS;
}
```

The dispatcher will find both methods called over the NSData instance and invoke both of them correctly.

Uninstantiable

Uninstantiable classes allow programmers to group together per-class fields and methods that are accessible at runtime without an instance of the class. Indeed, instantiation is prohibited for this kind of class.

For example, in C#, a class marked "static" can not be instantiated, can only have static members (fields, methods, other), may not have *instance constructors*, and is *sealed*.^[37]

Unnamed

An **unnamed class** or **anonymous class** is a class which is not bound to a name or identifier upon definition. This is analogous to named versus unnamed functions.

Benefits

Computer programs usually model aspects of some real or abstract world (the Domain). Because each class models a concept, classes provide a more natural way to create such models. Each class in the model represents a noun in the domain, and the methods of the class represent verbs that may apply to that noun (Verbs can also be modeled as classes, see Command Pattern). For example in a typical business system, various aspects of the business are modeled, using such classes as *Customer*, *Product*, *Worker*, *Invoice*, *Job*, etc. An *Invoice* may have methods like *Create*, *Print* or *Send*, a *Job* may be *Performed* or *Canceled*, etc. Once the system can model aspects of the business accurately, it can provide users of the system with useful information about those aspects. Classes allow a clear correspondence (mapping) between the model and the domain, making it easier to design, build, modify and

understand these models. Classes provide some control over the often challenging complexity of such models.

Classes can accelerate development by reducing redundant program code, testing and bug fixing. If a class has been thoroughly tested and is known to be a 'solid work', it is usually true that using or extending the well-tested class will reduce the number of bugs - as compared to the use of freshly developed or ad hoc code - in the final output. In addition, efficient class reuse means that many bugs need to be fixed in only one place when problems are discovered.

Another reason for using classes is to simplify the relationships of interrelated data. Rather than writing code to repeatedly call a graphical user interface (GUI) window drawing subroutine on the terminal screen (as would be typical for structured programming), it is more intuitive. With classes, GUI items that are similar to windows (such as dialog boxes) can simply inherit most of their functionality and data structures from the window class. The programmer then need only add code to the dialog class that is unique to its operation. Indeed, GUIs are a very common and useful application of classes, and GUI programming is generally much easier with a good class framework.

Run-time representation

As a data type, a class is usually considered as a compile-time construct. A language may also support prototype or factory metaobjects that represent run-time information about classes, or even represent metadata that provides access to reflection facilities and ability to manipulate data structure formats at run-time. Many languages distinguish this kind of run-time type information about classes from a class on the basis that the information is not needed at run-time. Some dynamic languages do not make strict distinctions between run-time and compile-time constructs, and therefore may not distinguish between metaobjects and classes.

For example, if Human is a metaobject representing the class Person, then instances of class Person can be created by using the facilities of the Human metaobject.

Notes

- [1] Gamma, Helm, Johnson, Vlissides 1995, p. 14
- [2] Gamma et al. 1995, p. 17
- [3] Gamma et al. 1995, p. 14
- [4] "3. Data model" (<http://docs.python.org/reference/datamodel.html>). *The Python Language Reference*. Python Software Foundation.. Retrieved 2012-04-26.
- [5] Booch 1994, p. 86-88
- [6] "Classes (I)" (<http://www.cplusplus.com/doc/tutorial/classes/>). *C++ Language Tutorial*. cplusplus.com.. Retrieved 2012-04-29.
- [7] "Classes (II)" (<http://www.cplusplus.com/doc/tutorial/classes2/>). *C++ Language Tutorial*. cplusplus.com.. Retrieved 2012-04-29.
- [8] Booch 1994, p. 105
- [9] "Controlling Access to Members of a Class" (<http://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html>). *The Java Tutorials*. Oracle.. Retrieved 2012-04-19.
- [10] "OOP35-CPP. Do not return references to private data" (<https://www.securecoding.cert.org/confluence/display/cplusplus/OOP35-CPP.+Do+not+return+references+to+private+data>). *CERT C++ Secure Coding Standard*. Carnegie Mellon University. 2010-05-10.. Retrieved 2012-05-07.
- [11] Ben-Ari, Mordechai (2007-01-24). "2.2 Identifiers" (<http://introcs.cs.princeton.edu/java/11cheatsheet/errors.pdf>). *Compile and Runtime Errors in Java*. . Retrieved 2012-05-07.
- [12] Wild, Fred. "C++ Interfaces" (<http://www.drdobbs.com/cpp/184410630>). *Dr. Dobb's*. UBM Techweb.. Retrieved 2012-05-02.
- [13] Thomas; Hunt. "Classes, Objects, and Variables" (http://ruby-doc.org/docs/ProgrammingRuby/html/tut_classes.html). *Programming Ruby: The Pragmatic Programmer's Guide*. Ruby-Doc.org.. Retrieved 2012-04-26.
- [14] "Friendship and inheritance" (<http://www.cplusplus.com/doc/tutorial/inheritance/>). *C++ Language Tutorial*. cplusplus.com.. Retrieved 2012-04-26.
- [15] Booch 1994, p. 180
- [16] Booch 1994, p. 128-129
- [17] Booch 1994, p. 112
- [18] "Interfaces" (<http://docs.oracle.com/javase/tutorial/java/IandI/createinterface.html>). *The Java Tutorials*. Oracle.. Retrieved 2012-05-01.

- [19] Bell, Donald. "UML Basics: The class diagram" (<http://www.ibm.com/developerworks/rational/library/content/RationalEdge/sep04/bell/>). *developer Works*. IBM. . Retrieved 2012-05-02.
- [20] Booch 1994, p. 179
- [21] "Polymorphism" (<http://www.cplusplus.com/doc/tutorial/polymorphism/>). *C++ Language Tutorial*. cplusplus.com. . Retrieved 2012-05-02.
- [22] "Abstract Methods and Classes" (<http://docs.oracle.com/javase/tutorial/java/IandI/abstract.html>). *The Java Tutorials*. Oracle. . Retrieved 2012-05-02.
- [23] "Class Abstraction" (<http://php.net/manual/en/language.oop5.abstract.php>). *PHP Manual*. The PHP Group. . Retrieved 2012-05-02.
- [24] "Interfaces (C# Programming Guide)". *C# Programming Guide*. Microsoft.
- [25] "Inheritance (C# Programming Guide)" (<http://msdn.microsoft.com/en-us/library/ms173149.aspx>). *C# Programming Guide*. Microsoft. . Retrieved 2012-05-02.
- [26] "Nested classes (C++ only)" (<http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=/com.ibm.xlcpp8a.doc/language/ref/cplr061.htm>). *XL C/C++ V8.0 for AIX*. IBM. . Retrieved 2012-05-07.
- [27] "Local type names (C++ only)" (<http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=/com.ibm.xlcpp8a.doc/language/ref/cplr063.htm>). *XL C/C++ V8.0 for AIX*. IBM. . Retrieved 2012-05-07.
- [28] "Local classes (C++ only)" (<http://publib.boulder.ibm.com/infocenter/comphelp/v8v101/index.jsp?topic=/com.ibm.xlcpp8a.doc/language/ref/cplr062.htm>). *XL C/C++ V8.0 for AIX*. IBM. . Retrieved 2012-05-07.
- [29] Booch 1994, p. 133-134
- [30] Thomas; Hunt. "Classes and Objects" (<http://www.ruby-doc.org/docs/ProgrammingRuby/html/classes.html>). *Programming Ruby: The Pragmatic Programmer's Guide*. Ruby-Doc.org. . Retrieved 2012-05-08.
- [31] Booch 1994, p. 134
- [32] "NSObject Class Reference" (https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/nsobject_Class/Reference/Reference.html). *Mac OS X Developer Library*. Apple. . Retrieved 2012-05-08.
- [33] "MOP: Concepts" (<http://www.alu.org/mop/concepts.html#introduction>). *The Common Lisp Object System MetaObject Protocol*. Association of Lisp Users. . Retrieved 2012-05-08.
- [34] "sealed (C# Reference)" (<http://msdn.microsoft.com/en-us/library/ms173149.aspx>). *C# Reference*. Microsoft. . Retrieved 2012-05-08.
- [35] "Writing Final Classes and Methods" (<http://docs.oracle.com/javase/tutorial/java/IandI/final.html>). *The Java Tutorials*. Oracle. . Retrieved 2012-05-08.
- [36] "String (Java Platform SE 7)" (<http://docs.oracle.com/javase/7/docs/api/java/lang/String.html>). *Java Platform, Standard Edition 7: API Specification*. Oracle. . Retrieved 2012-05-08.
- [37] "Static Classes and Static Class Members (C# Programming Guide)" ([http://msdn.microsoft.com/en-us/library/79b3xss3\(v=vs.100\).aspx](http://msdn.microsoft.com/en-us/library/79b3xss3(v=vs.100).aspx)). *C# Programming Guide*. Microsoft. . Retrieved 2012-05-08.

References

- Booch, Grady (1994). *Objects and Design with Applications, Second Edition*. Benjamin/Cummings.
- Gamma; Helm; Johnson; Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley.

Further reading

- Abadi; Cardelli: A Theory of Objects (<http://lucacardelli.name/TheoryOfObjects.html>)
- ISO/IEC 14882:2003 Programming Language C++, International standard (<http://www.open-std.org/jtc1/sc22/wg21/>)
- Class Warfare: Classes vs. Prototypes (<http://www.laputan.org/reflection/warfare.html>), by Brian Foote
- Meyer, B.: "Object-oriented software construction", 2nd edition, Prentice Hall, 1997, ISBN 0-13-629155-4
- Rumbaugh et al.: "Object-oriented modeling and design", Prentice Hall, 1991, ISBN 0-13-630054-5

External links

- Dias, Tiago (October 2006). "Programming demo - .NET using Partial Types for better code" (<http://www.youtube.com/watch?v=oaw8K8GNhAI>). *Hyper/Net*. Youtube.

Object (computer science)

In computer science, an **object** is a location in memory having a value and referenced by an identifier. An object can be a variable, function, or data structure. With the later introduction of object-oriented programming the same word, "object," refers to a particular instance of a class.

Object-oriented programming

Objects in "**object-oriented programming**" basically are data structures combined with the associated processing routines. For instance, a file is an object: a collection of data and the associated read and write routines. Objects are considered instantiations of **classes**. In common parlance one refers to *a* file as a class, while *the* file is the object. A class defines properties and behaviour once, usually for multiple instantiations. This distinction has its counterparts in other disciplines, for example in biology and evolution^[1] and is known as the *genus/species dichotomy*.

In the domain of object-oriented programming an object is usually taken to mean an ephemeral compilation of attributes (object elements) and behaviors (methods or subroutines) encapsulating an entity. In this way, while primitive or simple data types are still just single pieces of information, **object-oriented objects** are complex types that have multiple pieces of information and specific properties (or attributes). Instead of merely being assigned a value, (like int =10), objects have to be "constructed". In the real world, if a *Ford Focus* is an "object" - an instance of the *car* class, its physical properties and its function *to drive* would have been individually specified. Once the properties of the Ford Focus "object" had been specified into the form of the car class, it can be endlessly copied to create identical objects that look and function in just the same way. As an alternative example, animal is a superclass of primate and primate is a superclass of human. Individuals such as Joe Bloggs or John Doe would be particular examples or 'objects' of the human class, and consequently possess all the characteristics of the human class (and of the primate and animal superclasses as well).

"Objects" are the foundation of object-oriented programming, and are fundamental data types in object-oriented programming languages. These languages provide extensive syntactic and semantic support for object handling, including a hierarchical type system, special notation for declaring and calling methods, and facilities for hiding selected fields from client programmers. However, objects and object-oriented programming can be implemented in any language.

Objects are used in software development to implement abstract data structures, by bringing together the data components with the procedures that manipulate them. Objects in object-oriented programming are key in the concept of inheritance; thereby improving program reliability , simplification of software maintenance, the management of libraries, and the division of work in programmer teams. Object-oriented programming languages are generally designed to exploit and enforce these potential advantages of the object model. Objects can also make it possible to handle very disparate objects by the same piece of code, as long as they all have the proper method.

Properties of an object

Three properties characterize objects:

1. Identity: the property of an object that distinguishes it from other objects
2. State: describes the data stored in the object
3. Behavior: describes the methods in the object's interface by which the object can be used

Mechanism

The modern concept of "object" and the object-oriented approach to programming were introduced by the Simula programming language originally released in 1967, popularized by Smalltalk released two years later in 1969, and became standard tools of the trade with the spread of C++ originally released in 1983.

In the "pure" object-oriented approach, the data fields of an object should only be accessed through its methods (subroutines). It is claimed that this rule makes it easy to guarantee that the data will always remain in a valid state. Syntactically, in almost all object-oriented programming languages, a dot(.) operator (placed between an object and its symbolic method name) is used to call a particular method/function of an object. For example, consider an arithmetic class named Arith_Class. This class contains functions like add(), subtract(), multiply() and divide(), that process results for two numbers given to them. This class could be used to find the product of 78 and 69 by first creating an object of the class and then invoking its multiply method, as follows:

```
1 int result = 1;                                // Initialization
2 arith_Obj1 = new Arith_Class();                 // Creating a new object of Arith_Class
3 result = arith_Obj1.multiply(78,69);           // returned value of multiply function, store in result variable.
```

In a language where each object is created from a class, an object is called an **instance** of that class. If each object has a type, two objects with the same class would have the same data type. Creating an instance of a class is sometimes referred to as **instantiating** the class.

A real-world example of an object would be "my dog", which is an instance of a type (a class) called "dog", which is a subclass of a class "animal". In the case of a polymorphic object, some details of its type can be selectively ignored, for example a "dog" object could be used by a function looking for an "animal". So could a "cat", because it too belongs to the class of "animal". While being accessed as an "animal", some member attributes of a "dog" or "cat" would remain unavailable, such as the "tail" attribute, because not all animals have tails.

Specialized objects

There are certain specialized objects which can be created by the kind of Design Pattern which is used for creating them and usually those objects are named after their design patterns themselves. Some terms for specialized kinds of objects include

- Function object: an object with a single method (in C++, this method would be the function operator, "operator()") that acts much like a function (like a C/C++ pointer to a function).
- Immutable object: an object set up with a fixed state at creation time and which does not vary afterward.
- First-class object: an object that can be used without restriction.
- Container: an object that can contain other objects.
- Factory object: an object whose purpose is to create other objects.
- Metaobject: an object from which other objects can be created (Compare with class, which is not necessarily an object)
- Prototype: a specialized metaobject from which other objects can be created by copying
- God object: an object that knows *too much* or does *too much*. The God object is an example of an anti-pattern.
- Singleton object: An object that is the only instance of its class during the lifetime of the program.

- Filter object

In distributed computing

The definition of an object as an entity that has a distinct identity, state, and behavior, and, it is claimed, the principle of encapsulation, can be carried over to the realm of distributed computing. Paradoxically, encapsulation does not extend to an object's behavior since they (or even their method names) are not serialized along with the data. A number of extensions to the basic concept of an object have been proposed that share these common characteristics:

- *Distributed objects* are "ordinary" objects (objects in the usual sense - i.e. *not* OOP objects) that have been deployed at a number of distinct remote locations, and communicate by exchanging messages over the network. Examples include web services and DCOM objects.
- *Protocol objects* are components of a protocol stack that encapsulate network communication within an object-oriented interface.
- *Replicated objects* are groups of distributed objects (called *replicas*) that run a distributed multi-party protocol to achieve a high degree of consistency between their internal states, and that respond to requests in a coordinated manner. Referring to the group of replicas jointly as an *object* reflects the fact that interacting with any of them exposes the same externally visible state and behavior. Examples include fault-tolerant CORBA objects.
- *Live distributed objects* (or simply *live objects*)^[2] generalize the *replicated object* concept to groups of replicas that might internally use any distributed protocol, perhaps resulting in only a weak consistency between their local states.

Some of these extensions, such as *distributed objects* and *protocol objects*, are domain-specific terms for special types of "ordinary" objects used in a certain context (such as remote invocation or protocol composition). Others, such as *replicated objects* and *live distributed objects*, are more non-standard, in that they abandon the assumption that an object resides in a single location at a time, and apply the concept to groups of entities (replicas) that might span across multiple locations, might have only weakly consistent state, and whose membership might dynamically change.

Objects and the Semantic Web

It is claimed that the Semantic Web can be seen as a distributed data objects framework, and can therefore be validly seen as an object-oriented framework.^{[3][4]} However, a distributed object is regarded as an "ordinary" object, and not an OOP object because it is separated from its methods with which it was previously encapsulated. It is also claimed that it is valid to use a UML diagram to express a Semantic Web graph.

Both the Semantic Web and object-oriented programming have:

- Classes
- Attributes (also known as Relationships)
- Instances

Furthering this, Linked Data also introduces Dereferenceable Uniform Resource Identifiers, which provide data-by-reference which is found in object-oriented programming and object-oriented databases in the form of object identifiers.

References

- [1] Wildlife - Island Life - Darwin's Finches, Educational workbook, Imperial College London retrieved 29 June 2012, <http://www.doc.ic.ac.uk/~kpt/terraquest/galapagos/wildlife/island/finch.html>
- [2] Ostrowski, K., Birman, K., Dolev, D., and Ahnn, J. (2008). "Programming with Live Distributed Objects", *Proceedings of the 22nd European Conference on Object-Oriented Programming*, Paphos, Cyprus, July 07 - 11, 2008, J. Vitek, Ed., *Lecture Notes In Computer Science*, vol. 5142, Springer-Verlag, Berlin, Heidelberg, 463-489, <http://portal.acm.org/citation.cfm?id=1428508.1428536>.
- [3] Knublauch, Holger; Oberle, Daniel; Tetlow, Phil; Evan (2006-03-09). "A Semantic Web Primer for Object-Oriented Software Developers" (<http://www.w3.org/2001/sw/BestPractices/SE/ODSD/>). W3C. . Retrieved 2008-07-30.
- [4] Connolly, Daniel (2002-08-13). "An Evaluation of the World Wide Web with respect to Engelbart's Requirements" (<http://www.w3.org/Architecture/NOTE-ioh-arch>). W3C. . Retrieved 2008-07-30.

External links

- *What Is an Object?* (<http://java.sun.com/docs/books/tutorial/java/concepts/object.html>) from *The Java Tutorials*

Object-oriented programming

Object-oriented programming (OOP) is a programming paradigm using "objects" – usually instances of a class – consisting of data fields and methods together with their interactions – to design applications and computer programs. Programming techniques may include features such as data abstraction, encapsulation, messaging, modularity, polymorphism, and inheritance. Many modern programming languages now support forms of OOP, at least as an option.

Overview

An object-oriented program may be viewed as a collection of interacting *objects*, as opposed to the conventional model, in which a program is seen as a list of tasks (subroutines) to perform. In OOP, each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent "machine" with a distinct role or responsibility. The actions (or "methods") on these objects are closely associated with the object. For example, OOP data structures tend to "carry their own operators around with them" (or at least "inherit" them from a similar object or class) - except when they have to be serialized.

Simple, non-OOP programs may be one "long" list of statements (or commands). More complex programs will often group smaller sections of these statements into functions or subroutines each of which might perform a particular task. With designs of this sort, it is common for some of the program's data to be 'global', i.e. accessible from any part of the program. As programs grow in size, allowing any function to modify any piece of data means that bugs can have wide-reaching effects.

In contrast, the object-oriented approach encourages the programmer to place data where it is not directly accessible by the rest of the program. Instead, the data is accessed by calling specially written functions, commonly called methods, which are either bundled in with the data or inherited from "class objects." These act as the intermediaries for retrieving or modifying the data they control. The programming construct that combines data with a set of methods for accessing and managing those data is called an object. The practice of using subroutines to examine or modify certain kinds of data was also used in non-OOP modular programming, well before the widespread use of object-oriented programming.

An object-oriented program will usually contain different types of objects, each type corresponding to a particular kind of complex data to be managed or perhaps to a real-world object or concept such as a bank account, a hockey player, or a bulldozer. A program might well contain multiple copies of each type of object, one for each of the real-world objects the program is dealing with. For instance, there could be one bank account object for each

real-world account at a particular bank. Each copy of the bank account object would be alike in the methods it offers for manipulating or reading its data, but the data inside each object would differ reflecting the different history of each account.

Objects can be thought of as wrapping their data within a set of functions designed to ensure that the data are used appropriately, and to assist in that use. The object's methods will typically include checks and safeguards that are specific to the types of data the object contains. An object can also offer simple-to-use, standardized methods for performing particular operations on its data, while concealing the specifics of how those tasks are accomplished. In this way alterations can be made to the internal structure or methods of an object without requiring that the rest of the program be modified. This approach can also be used to offer standardized methods across different types of objects. As an example, several different types of objects might offer print methods. Each type of object might implement that print method in a different way, reflecting the different kinds of data each contains, but all the different print methods might be called in the same standardized manner from elsewhere in the program. These features become especially useful when more than one programmer is contributing code to a project or when the goal is to reuse code between projects.

Object-oriented programming has roots that can be traced to the 1960s. As hardware and software became increasingly complex, manageability often became a concern. Researchers studied ways to maintain software quality and developed object-oriented programming in part to address common problems by strongly emphasizing discrete, reusable units of programming logic. The technology focuses on data rather than processes, with programs composed of self-sufficient modules ("classes"), each instance of which ("objects") contains all the information needed to manipulate its own data structure ("members"). This is in contrast to the existing modular programming that had been dominant for many years that focused on the *function* of a module, rather than specifically the data, but equally provided for code reuse, and self-sufficient reusable units of programming logic, enabling collaboration through the use of linked modules (subroutines).

History

The terms "objects" and "oriented" in something like the modern sense of object-oriented programming seem to make their first appearance at MIT in the late 1950s and early 1960s. In the environment of the artificial intelligence group, as early as 1960, "object" could refer to identified items (LISP atoms) with properties (attributes);^{[1][2]} Alan Kay was later to cite a detailed understanding of LISP internals as a strong influence on his thinking in 1966.^[3] Another early MIT example was Sketchpad created by Ivan Sutherland in 1960–61; in the glossary of the 1963 technical report based on his dissertation about Sketchpad, Sutherland defined notions of "object" and "instance" (with the class concept covered by "master" or "definition"), albeit specialized to graphical interaction.^[4] Also, an MIT ALGOL version, AED-0, linked data structures ("plexes", in that dialect) directly with procedures, prefiguring what were later termed "messages", "methods" and "member functions".^{[5][6]}

Objects as a formal concept in programming were introduced in the 1960s in Simula 67, a major revision of Simula I, a programming language designed for discrete event simulation, created by Ole-Johan Dahl and Kristen Nygaard of the Norwegian Computing Center in Oslo.^[7] Simula 67 was influenced by SIMSCRIPT and C.A.R. "Tony" Hoare's proposed "record classes".^{[5][8]} Simula introduced the notion of classes and instances or objects (as well as subclasses, virtual methods, coroutines, and discrete event simulation) as part of an explicit programming paradigm. The language also used automatic garbage collection that had been invented earlier for the functional programming language Lisp. Simula was used for physical modeling, such as models to study and improve the movement of ships and their content through cargo ports. The ideas of Simula 67 influenced many later languages, including Smalltalk, derivatives of LISP (CLOS), Object Pascal, and C++.

The Smalltalk language, which was developed at Xerox PARC (by Alan Kay and others) in the 1970s, introduced the term *object-oriented programming* to represent the pervasive use of objects and messages as the basis for computation. Smalltalk creators were influenced by the ideas introduced in Simula 67, but Smalltalk was designed to

be a fully dynamic system in which classes could be created and modified dynamically rather than statically as in Simula 67.^[9] Smalltalk and with it OOP were introduced to a wider audience by the August 1981 issue of *Byte Magazine*.

In the 1970s, Kay's Smalltalk work had influenced the Lisp community to incorporate object-based techniques that were introduced to developers via the Lisp machine. Experimentation with various extensions to Lisp (like LOOPS and Flavors introducing multiple inheritance and mixins), eventually led to the Common Lisp Object System (CLOS, a part of the first standardized object-oriented programming language, ANSI Common Lisp), which integrates functional programming and object-oriented programming and allows extension via a Meta-object protocol. In the 1980s, there were a few attempts to design processor architectures that included hardware support for objects in memory but these were not successful. Examples include the Intel iAPX 432 and the Linn Smart Rekursiv.

In 1985, Bertrand Meyer produced the first design of the Eiffel language. Focused on software quality, Eiffel is among the purely object-oriented languages, but differs in the sense that the language itself is not only a programming language, but a notation supporting the entire software lifecycle. Meyer described the Eiffel software development method, based on a small number of key ideas from software engineering and computer science, in *Object-Oriented Software Construction*. Essential to the quality focus of Eiffel is Meyer's reliability mechanism, Design by Contract, which is an integral part of both the method and language.

Object-oriented programming developed as the dominant programming methodology in the early and mid 1990s when programming languages supporting the techniques became widely available. These included Visual FoxPro 3.0,^{[10][11][12]} C++, and Delphi. Its dominance was further enhanced by the rising popularity of graphical user interfaces, which rely heavily upon object-oriented programming techniques. An example of a closely related dynamic GUI library and OOP language can be found in the Cocoa frameworks on Mac OS X, written in Objective-C, an object-oriented, dynamic messaging extension to C based on Smalltalk. OOP toolkits also enhanced the popularity of event-driven programming (although this concept is not limited to OOP). Some feel that association with GUIs (real or perceived) was what propelled OOP into the programming mainstream.

At ETH Zürich, Niklaus Wirth and his colleagues had also been investigating such topics as data abstraction and modular programming (although this had been in common use in the 1960s or earlier). Modula-2 (1978) included both, and their succeeding design, Oberon, included a distinctive approach to object orientation, classes, and such. The approach is unlike Smalltalk, and very unlike C++.

Object-oriented features have been added to many existing languages during that time, including Ada, BASIC, Fortran, Pascal, and others. Adding these features to languages that were not initially designed for them often led to problems with compatibility and maintainability of code.

More recently, a number of languages have emerged that are primarily object-oriented yet compatible with procedural methodology, such as Python and Ruby. Probably the most commercially important recent object-oriented languages are Visual Basic.NET (VB.NET) and C#, both designed for Microsoft's .NET platform, and Java, developed by Sun Microsystems. Both frameworks show the benefit of using OOP by creating an abstraction from implementation in their own way. VB.NET and C# support cross-language inheritance, allowing classes defined in one language to subclass classes defined in the other language. Developers usually compile Java to bytecode, allowing Java to run on any operating system for which a Java virtual machine is available. VB.NET and C# make use of the Strategy pattern to accomplish cross-language inheritance, whereas Java makes use of the Adapter pattern.

Just as procedural programming led to refinements of techniques such as structured programming, modern object-oriented software design methods include refinements such as the use of design patterns, design by contract, and modeling languages (such as UML).

Fundamental features and concepts

A survey by Deborah J. Armstrong of nearly 40 years of computing literature identified a number of "quarks", or fundamental concepts, found in the strong majority of definitions of OOP.^[13]

Not all of these concepts are to be found in all object-oriented programming languages. For example, object-oriented programming that uses classes is sometimes called class-based programming, while prototype-based programming does not typically use classes. As a result, a significantly different yet analogous terminology is used to define the concepts of *object* and *instance*.

Benjamin C. Pierce and some other researchers view as futile any attempt to distill OOP to a minimal set of features. He nonetheless identifies fundamental features that support the OOP programming style in most object-oriented languages:^[14]

- Dynamic dispatch – when a method is invoked on an object, the object itself determines what code gets executed by looking up the method at run time in a table associated with the object. This feature distinguishes an object from an abstract data type (or module), which has a fixed (static) implementation of the operations for all instances. It is a programming methodology that gives modular component development while at the same time being very efficient.
- Encapsulation (or multi-methods, in which case the state is kept separate)
- Subtype polymorphism
- Object inheritance (or delegation)
- Open recursion – a special variable (syntactically it may be a keyword), usually called `this` or `self`, that allows a method body to invoke another method body of the same object. This variable is *late-bound*; it allows a method defined in one class to invoke another method that is defined later, in some subclass thereof.

Similarly, in his 2003 book, *Concepts in programming languages*, John C. Mitchell identifies four main features: dynamic dispatch, abstraction, subtype polymorphism, and inheritance.^[15] Michael Lee Scott in *Programming Language Pragmatics* considers only encapsulation, inheritance and dynamic dispatch.^[16]

Additional concepts used in object-oriented programming include:

- Classes of objects
- Instances of classes
- Methods which act on the attached objects.
- Message passing
- Abstraction

Decoupling

Decoupling refers to careful controls that separate code modules from particular use cases, which increases code re-usability. A common use of decoupling in OOP is to polymorphically decouple the encapsulation (see Bridge pattern and Adapter pattern) - for example, using a method interface which an encapsulated object must satisfy, as opposed to using the object's class.

Additional Features

1. Encapsulation Enforces Modularity: Encapsulation refers to the creation of self-contained modules that bind processing functions to the data. These user-defined data types are called "classes," and one instance of a class is an "object." For example, in a payroll system, a class could be Manager, and Pat and Jan could be two instances (two objects) of the Manager class. Encapsulation ensures good code modularity, which keeps routines separate and less prone to conflict with each other.

2. Inheritance Passes "Knowledge" Down: Classes are created in hierarchies, and inheritance allows the structure and methods in one class to be passed down the hierarchy. That means less programming is required when adding

functions to complex systems. If a step is added at the bottom of a hierarchy, then only the processing and data associated with that unique step needs to be added. Everything else about that step is inherited. The ability to reuse existing objects is considered a major advantage of object technology.

3. Polymorphism Takes any Shape: Object-oriented programming allows procedures about objects to be created whose exact type is not known until runtime. For example, a screen cursor may change its shape from an arrow to a line depending on the program mode. The routine to move the cursor on screen in response to mouse movement would be written for "cursor," and polymorphism allows that cursor to take on whatever shape is required at runtime. It also allows new shapes to be easily integrated.

4. OOP Languages: Used for simulating system behavior in the late 1960s, SIMULA was the first object-oriented language. In the 1970s, Xerox's Smalltalk was the first object-oriented programming language and was used to create the graphical user interface (GUI). Today, C++ and Java are the major OOP languages, while C#, Visual Basic.NET, Python and JavaScript are also popular. ACTOR and Eiffel were earlier OOP languages. The following list compares some basic OOP terms with traditional programming.

Formal semantics

There have been several attempts at formalizing the concepts used in object-oriented programming. The following concepts and constructs have been used as interpretations of OOP concepts:

- coalgebraic data types [17]
- abstract data types (which have existential types) allow the definition of modules but these do not support dynamic dispatch
- recursive types
- encapsulated state
- Inheritance
- records are basis for understanding objects if function literals can be stored in fields (like in functional programming languages), but the actual calculi need be considerably more complex to incorporate essential features of OOP. Several extensions of System F_< that deal with mutable objects have been studied; [18] these allow both subtype polymorphism and parametric polymorphism (generics)

Attempts to find a consensus definition or theory behind objects have not proven very successful (however, see Abadi & Cardelli, *A Theory of Objects* [19][18] for formal definitions of many OOP concepts and constructs), and often diverge widely. For example, some definitions focus on mental activities, and some on program structuring. One of the simpler definitions is that OOP is the act of using "map" data structures or arrays that can contain functions and pointers to other maps, all with some syntactic and scoping sugar on top. Inheritance can be performed by cloning the maps (sometimes called "prototyping"). OBJECT:=>> Objects are the run time entities in an object-oriented system. They may represent a person, a place, a bank account, a table of data or any item that the program has to handle.

OOP languages

Simula (1967) is generally accepted as the first language to have the primary features of an object-oriented language. It was created for making simulation programs, in which what came to be called objects were the most important information representation. Smalltalk (1972 to 1980) is arguably the canonical example, and the one with which much of the theory of object-oriented programming was developed. Concerning the degree of object orientation, the following distinctions can be made:

- Languages called "pure" OO languages, because everything in them is treated consistently as an object, from primitives such as characters and punctuation, all the way up to whole classes, prototypes, blocks, modules, etc. They were designed specifically to facilitate, even enforce, OO methods. Examples: Eiffel, Emerald.^[20], JADE,

Obix, Scala, Smalltalk

- Languages designed mainly for OO programming, but with some procedural elements. Examples: C++, Java, C#, VB.NET, Python.
- Languages that are historically procedural languages, but have been extended with some OO features. Examples: Visual Basic (derived from BASIC), Fortran, Perl, COBOL 2002, PHP, ABAP.
- Languages with most of the features of objects (classes, methods, inheritance, reusability), but in a distinctly original form. Examples: Oberon (Oberon-1 or Oberon-2) and Common Lisp.
- Languages with abstract data type support, but not all features of object-orientation, sometimes called *object-based* languages. Examples: Modula-2, Pliant, CLU.

OOP in dynamic languages

In recent years, object-oriented programming has become especially popular in dynamic programming languages. Python, Ruby and Groovy are dynamic languages built on OOP principles, while Perl and PHP have been adding object oriented features since Perl 5 and PHP 4, and ColdFusion since version 5.

The Document Object Model of HTML, XHTML, and XML documents on the Internet have bindings to the popular JavaScript/ECMAScript language. JavaScript is perhaps the best known prototype-based programming language, which employs cloning from prototypes rather than inheriting from a class. Another scripting language that takes this approach is Lua. Earlier versions of ActionScript (a partial superset of the ECMA-262 R3, otherwise known as ECMAScript) also used a prototype-based object model. Later versions of ActionScript incorporate a combination of classification and prototype-based object models based largely on the currently incomplete ECMA-262 R4 specification, which has its roots in an early JavaScript 2 Proposal. Microsoft's JScript.NET also includes a mash-up of object models based on the same proposal, and is also a superset of the ECMA-262 R3 specification.

Design patterns

Challenges of object-oriented design are addressed by several methodologies. Most common is known as the design patterns codified by Gamma *et al.*. More broadly, the term "design patterns" can be used to refer to any general, repeatable solution to a commonly occurring problem in software design. Some of these commonly occurring problems have implications and solutions particular to object-oriented development.

Inheritance and behavioral subtyping

It is intuitive to assume that inheritance creates a semantic "is a" relationship, and thus to infer that objects instantiated from subclasses can always be *safely* used instead of those instantiated from the superclass. This intuition is unfortunately false in most OOP languages, in particular in all those that allow mutable objects. Subtype polymorphism as enforced by the type checker in OOP languages (with mutable objects) cannot guarantee behavioral subtyping in any context. Behavioral subtyping is undecidable in general, so it cannot be implemented by a program (compiler). Class or object hierarchies need to be carefully designed considering possible incorrect uses that cannot be detected syntactically. This issue is known as the Liskov substitution principle.

Gang of Four design patterns

Design Patterns: Elements of Reusable Object-Oriented Software is an influential book published in 1995 by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, often referred to humorously as the "Gang of Four". Along with exploring the capabilities and pitfalls of object-oriented programming, it describes 23 common programming problems and patterns for solving them. As of April 2007, the book was in its 36th printing.

The book describes the following patterns:

- *Creational patterns* (5): Factory method pattern, Abstract factory pattern, Singleton pattern, Builder pattern, Prototype pattern
- *Structural patterns* (7): Adapter pattern, Bridge pattern, Composite pattern, Decorator pattern, Facade pattern, Flyweight pattern, Proxy pattern
- *Behavioral patterns* (11): Chain-of-responsibility pattern, Command pattern, Interpreter pattern, Iterator pattern, Mediator pattern, Memento pattern, Observer pattern, State pattern, Strategy pattern, Template method pattern, Visitor pattern...

Object-orientation and databases

Both object-oriented programming and relational database management systems (RDBMSs) are extremely common in software today. Since relational databases don't store objects directly (though some RDBMSs have object-oriented features to approximate this), there is a general need to bridge the two worlds. The problem of bridging object-oriented programming accesses and data patterns with relational databases is known as Object-Relational impedance mismatch. There are a number of approaches to cope with this problem, but no general solution without downsides.^[21] One of the most common approaches is object-relational mapping, as found in libraries like Java Data Objects and Ruby on Rails' ActiveRecord.

There are also object databases that can be used to replace RDBMSs, but these have not been as technically and commercially successful as RDBMSs.

Real-world modeling and relationships

OOP can be used to associate real-world objects and processes with digital counterparts. However, not everyone agrees that OOP facilitates direct real-world mapping (see Negative Criticism section) or that real-world mapping is even a worthy goal; Bertrand Meyer argues in *Object-Oriented Software Construction*^[22] that a program is not a model of the world but a model of some part of the world; "Reality is a cousin twice removed". At the same time, some principal limitations of OOP had been noted.^[23] For example, the Circle-ellipse problem is difficult to handle using OOP's concept of inheritance.

However, Niklaus Wirth (who popularized the adage now known as Wirth's law: "Software is getting slower more rapidly than hardware becomes faster") said of OOP in his paper, "Good Ideas through the Looking Glass", "This paradigm closely reflects the structure of systems 'in the real world', and it is therefore well suited to model complex systems with complex behaviours" (contrast KISS principle).

Steve Yegge and others noted that natural languages lack the OOP approach of strictly prioritizing *things* (objects/nouns) before *actions* (methods/verbs).^[24] This problem may cause OOP to suffer more convoluted solutions than procedural programming.^[25]

OOP and control flow

OOP was developed to increase the reusability and maintainability of source code.^[26] Transparent representation of the control flow had no priority and was meant to be handled by a compiler. With the increasing relevance of parallel hardware and multithreaded coding, developer transparent control flow becomes more important, something hard to achieve with OOP.^{[27][28][29][30]}

Responsibility- vs. data-driven design

Responsibility-driven design defines classes in terms of a contract, that is, a class should be defined around a responsibility and the information that it shares. This is contrasted by Wirfs-Brock and Wilkerson with data-driven design, where classes are defined around the data-structures that must be held. The authors hold that responsibility-driven design is preferable.

Criticism

A number of well-known researchers and programmers have analysed the utility of OOP. Here is an incomplete list:

- Luca Cardelli wrote a paper titled "Bad Engineering Properties of Object-Oriented Languages".^[31]
- Richard Stallman wrote in 1995, "Adding OOP to Emacs is not clearly an improvement; I used OOP when working on the Lisp Machine window systems, and I disagree with the usual view that it is a superior way to program."^[32]
- A study by Potok et al.^[33] has shown no significant difference in productivity between OOP and procedural approaches.
- Christopher J. Date stated that critical comparison of OOP to other technologies, relational in particular, is difficult because of lack of an agreed-upon and rigorous definition of OOP.^[34] Date and Darwen^[35] propose a theoretical foundation on OOP that uses OOP as a kind of customizable type system to support RDBMS.
- Alexander Stepanov suggested that OOP provides a mathematically limited viewpoint and called it "almost as much of a hoax as Artificial Intelligence. I have yet to see an interesting piece of code that comes from these OO people. In a sense, I am unfair to AI: I learned a lot of stuff from the MIT AI Lab crowd, they have done some really fundamental work....".^[36]
- Paul Graham has suggested that the purpose of OOP is to act as a "herding mechanism" that keeps mediocre programmers in mediocre organizations from "doing too much damage". This is at the expense of slowing down productive programmers who know how to use more powerful and more compact techniques.^[37]
- Joe Armstrong, the principal inventor of Erlang, is quoted as saying "The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."^[38]
- Richard Mansfield, author and former editor of *COMPUTE!* magazine, states that "like countless other intellectual fads over the years ("relevance", communism, "modernism", and so on—history is littered with them), OOP will be with us until eventually reality asserts itself. But considering how OOP currently pervades both universities and workplaces, OOP may well prove to be a durable delusion. Entire generations of indoctrinated programmers continue to march out of the academy, committed to OOP and nothing but OOP for the rest of their lives."^[39] He also is quoted as saying "OOP is to writing a program, what going through airport security is to flying".^[40]
- Steve Yegge, making a roundabout comparison with Functional programming, writes, "Object Oriented Programming puts the Nouns first and foremost. Why would you go to such lengths to put one part of speech on a pedestal? Why should one kind of concept take precedence over another? It's not as if OOP has suddenly made verbs less important in the way we actually think. It's a strangely skewed perspective."^[41]
- Rich Hickey, creator of Clojure, described object systems as over simplistic models of the real world. He emphasized the inability of OOP to model time properly, which is getting increasingly problematic as software

- systems become more concurrent.^[42]
- Carnegie-Mellon University Professor Robert Harper in March 2011 wrote: "This semester Dan Licata and I are co-teaching a new course on functional programming for first-year prospective CS majors... Object-oriented programming is eliminated entirely from the introductory curriculum, because it is both anti-modular and anti-parallel by its very nature, and hence unsuitable for a modern CS curriculum. A proposed new course on object-oriented design methodology will be offered at the sophomore level for those students who wish to study this topic."^[43]

References

- [1] McCarthy, J.; Brayton, R.; Edwards, D.; Fox, P.; Hodes, L.; Luckham, D.; Maling, K.; Park, D. et al. (March 1960). *LISP I Programmers Manual* (http://history.siam.org/sup/Fox_1960_LISP.pdf). Boston, Massachusetts: Artificial Intelligence Group, M.I.T. Computation Center and Research Laboratory. p. 88f. . "In the local M.I.T. patois, association lists [of atomic symbols] are also referred to as "property lists", and atomic symbols are sometimes called "objects"."
- [2] McCarthy, John; Abrahams, Paul W.; Edwards, Daniel J.; Hart, swapnil d.; Levin, Michael I. (1962). *LISP 1.5 Programmer's Manual* (<http://community.computerhistory.org/scc/projects/LISP/book/LISP 1.5 Programmers Manual.pdf>). MIT Press. p. 105. ISBN 0-262-13011-4. . "Object - a synonym for atomic symbol"
- [3] "Dr. Alan Kay on the Meaning of "Object-Oriented Programming"" (http://www.purl.org/stefan_ram/pub/doc_kay_oop_en). 2003. . Retrieved 11 February 2010.
- [4] Sutherland, I. E. (30 January 1963). "Sketchpad: A Man-Machine Graphical Communication System" (<http://handle.dtic.mil/100.2/AD404549>) (PDF). Technical Report No. 296, Lincoln Laboratory, Massachusetts Institute of Technology via Defense Technical Information Center (stinet.dtic.mil). . Retrieved 3 November 2007.
- [5] The Development of the Simula Languages, Kristen Nygaard, Ole-Johan Dahl, p.254 Uni-kl.ac.at (http://cs-exhibitions.uni-klu.ac.at/fileadmin/template/documents/text/The_development_of_the_simula_languages.pdf)
- [6] Ross, Doug. "The first software engineering language" (<http://www.csail.mit.edu/timeline/timeline.php?query=event&id=19>). *LCS/AI Lab Timeline*: MIT Computer Science and Artificial Intelligence Laboratory. . Retrieved 13 May 2010.
- [7] Holmevik, Jan Rune (1994). "Compiling Simula: A historical study of technological genesis" (<http://www.idi.ntnu.no/grupper/su/publications/holmevik-simula-ieeeannals94.pdf>). *IEEE Annals in the History of Computing* **16** (4): 25–37. doi:10.1109/85.329756. . Retrieved 12 May 2010.
- [8] Hoare, C. A. (Nov 1965). "Record Handling". *ALGOL Bulletin* (21): 39–69. doi:10.1145/1061032.1061041
- [9] Kay, Alan. "The Early History of Smalltalk" (<http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html>). . Retrieved 13 September 2007.
- [10] 1995 (June) Visual FoxPro 3.0, FoxPro evolves from a procedural language to an object-oriented language. Visual FoxPro 3.0 introduces a database container, seamless client/server capabilities, support for ActiveX technologies, and OLE Automation and null support. Summary of Fox releases (http://www.foxprohistory.org/foxprotimeline.htm#summary_of_fox_releases)
- [11] FoxPro History web site: Foxprohistory.org (<http://www.foxprohistory.org/tableofcontents.htm>)
- [12] 1995 Reviewers Guide to Visual FoxPro 3.0: DFpug.de (http://www.dfpug.de/loseblattsammlung/migration/whitepapers/vfp_rg.htm)
- [13] Armstrong, *The Quarks of Object-Oriented Development*. In descending order of popularity, the "quarks" are: Inheritance, Object, Class, Encapsulation, Method, Message Passing, Polymorphism, Abstraction
- [14] Pierce, Benjamin (2002). *Types and Programming Languages*. MIT Press. ISBN 0-262-16209-1., section 18.1 "What is Object-Oriented Programming?"
- [15] John C. Mitchell, *Concepts in programming languages*, Cambridge University Press, 2003, ISBN 0-521-78098-5, p.278
- [16] Michael Lee Scott, *Programming language pragmatics*, Edition 2, Morgan Kaufmann, 2006, ISBN 0-12-633951-1, p. 470 vikas
- [17] Poll, Erik. "Subtyping and Inheritance for Categorical Datatypes" (<http://www.cs.ru.nl/E.Poll/papers/kyoto97.pdf>). . Retrieved 5 June 2011.
- [18] Abadi, Martin; Cardelli, Luca (1996). *A Theory of Objects* (<http://portal.acm.org/citation.cfm?id=547964&dl=ACM&coll=portal>). Springer-Verlag New York, Inc.. ISBN 0-387-94775-2. . Retrieved 21 April 2010.
- [19] <http://portal.acm.org/citation.cfm?id=547964&dl=ACM&coll=portal>
- [20] "The Emerald Programming Language" (<http://www.emeraldprogramminglanguage.org/>). .
- [21] Neward, Ted (26 June 2006). "The Vietnam of Computer Science" (<http://blogs.tedneward.com/2006/06/26/The+Vietnam+Of+Computer+Science.aspx>). Interoperability Happens. . Retrieved 2 June 2010.
- [22] Meyer, Second Edition, p. 230
- [23] M.Trofimov, *OOOP - The Third "O" Solution: Open OOP*. First Class, OMG, 1993, Vol. 3, issue 3, p.14.
- [24] Yegge, Steve (30 March 2006). "Execution in the Kingdom of Nouns" (<http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>). steve-yegge.blogspot.com. . Retrieved 3 July 2010.
- [25] Boronczyk, Timothy (11 June 2009). "What's Wrong with OOP" (<http://zaemis.blogspot.com/2009/06/whats-wrong-with-oop.html>). zaemis.blogspot.com. . Retrieved 3 July 2010.

- [26] Ambler, Scott (1 January 1998). "A Realistic Look at Object-Oriented Reuse" (<http://www.drdobbs.com/184415594>). www.drdobbs.com. . Retrieved 4 July 2010.
- [27] Shelly, Asaf (22 August 2008). "Flaws of Object Oriented Modeling" (<http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/>). Intel Software Network. . Retrieved 4 July 2010.
- [28] James, Justin (1 October 2007). "Multithreading is a verb not a noun" (<http://blogs.techrepublic.com.com/programming-and-development/?p=518>). techrepublic.com. . Retrieved 4 July 2010.
- [29] Shelly, Asaf (22 August 2008). "HOW TO: Multicore Programming (Multiprocessing) Visual C++ Class Design Guidelines, Member Functions" (<http://support.microsoft.com/?scid=kb;en-us;558117>). support.microsoft.com. . Retrieved 4 July 2010.
- [30] Robert Harper (17 April 2011). "Some thoughts on teaching FP" (<http://existentialtype.wordpress.com/2011/04/17/some-advice-on-teaching-fp/>). Existential Type Blog. . Retrieved 5 December 2011.
- [31] Cardelli, Luca (1996). "Bad Engineering Properties of Object-Oriented Languages" (<http://lucacardelli.name/Papers/BadPropertiesOfOO.html>). ACM Comput. Surv. (ACM) **28** (4es): 150. doi:10.1145/242224.242415. ISSN 0360-0300. . Retrieved 21 April 2010.
- [32] Stallman, Richard (16 January 1995). "Mode inheritance, cloning, hooks & OOP" (http://groups.google.com/group/comp.emacs.xemacs/browse_thread/thread/d0af257a2837640c/37f251537fafbb03?lnk=st&q=%22Richard+Stallman%22+oop&rnum=5&hl=en#37f251537fafbb03). Google Groups Discussion. . Retrieved 21 June 2008.
- [33] Potok, Thomas; Mladen Vouk, Andy Rindos (1999). "Productivity Analysis of Object-Oriented Software Developed in a Commercial Environment" (<http://www.csm.ornl.gov/~v8q/Homepage/Papers Old/spetep- printable.pdf>). Software – Practice and Experience **29** (10): 833–847. doi:10.1002/(SICI)1097-024X(199908)29:10<833::AID-SPE258>3.0.CO;2-P. . Retrieved 21 April 2010.
- [34] C. J. Date, Introduction to Database Systems, 6th-ed., Page 650
- [35] C. J. Date, Hugh Darwen, *Foundation for Future Database Systems: The Third Manifesto* (2nd Edition)
- [36] Stepanov, Alexander. "STLport: An Interview with A. Stepanov" (<http://www.stlport.org/resources/StepanovUSA.html>). . Retrieved 21 April 2010.
- [37] Graham, Paul. "Why ARC isn't especially Object–Oriented." (<http://www.paulgraham.com/noop.html>). PaulGraham.com. . Retrieved 13 November 2009.
- [38] Armstrong, Joe. In *Coders at Work: Reflections on the Craft of Programming*. Peter Seibel, ed. Codersatwork.com (<http://www.codersatwork.com/>), Accessed 13 November 2009.
- [39] Mansfield, Richard. "Has OOP Failed?" 2005. Available at 4JS.com (http://www.4js.com/en/fichiers/b_genero/pourquoi/Has_OOP_Failed_Sept_2005.pdf), Accessed 13 November 2009.
- [40] Mansfield, Richard. "OOP Is Much Better in Theory Than in Practice" 2005. Available at Devx.com (<http://www.devx.com/DevX/Article/26776>) Accessed 7 January 2010.
- [41] Stevey's Blog Rants (<http://steve-yegge.blogspot.com/2006/03/execution-in-kingdom-of-nouns.html>)
- [42] Rich Hickey, JVM Languages Summit 2009 keynote, Are We There Yet? (<http://www.infoq.com/presentations/Are-We-There-Yet-Rich-Hickey>) November 2009.
- [43] Teaching FP to Freshmen (<http://existentialtype.wordpress.com/2011/03/15/teaching-fp-to-freshmen/>), from Harper's blog about teaching introductory computer science. (<http://existentialtype.wordpress.com/2011/03/15/getting-started/>)

Further reading

- [Weisfeld, Matt (<http://www.linkedin.com/in/mattweisfeld>)] (2009). *The Object-Oriented Thought Process, Third Edition*. Addison-Wesley. ISBN 0-672-33016-4.
- Schach, Stephen (2006). *Object-Oriented and Classical Software Engineering, Seventh Edition*. McGraw-Hill. ISBN 0-07-319126-4.
- Abadi, Martin; Luca Cardelli (1998). *A Theory of Objects*. Springer Verlag. ISBN 0-387-94775-2.
- Abelson, Harold; Gerald Jay Sussman, (1997). *Structure and Interpretation of Computer Programs* (<http://mitpress.mit.edu/sicp/>). MIT Press. ISBN 0-262-01153-0.
- Armstrong, Deborah J. (February 2006). "The Quarks of Object-Oriented Development" (<http://portal.acm.org/citation.cfm?id=1113040>). *Communications of the ACM* **49** (2): 123–128. doi:10.1145/1113034.1113040. ISSN 0001-0782. Retrieved 8 August 2006.
- Booch, Grady (1997). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley. ISBN 0-8053-5340-2.
- Eeles, Peter; Oliver Sims (1998). *Building Business Objects*. John Wiley & Sons. ISBN 0-471-19176-0.
- Gamma, Erich; Richard Helm, Ralph Johnson, John Vlissides (1995). *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.
- Harmon, Paul; William Morrissey (1996). *The Object Technology Casebook - Lessons from Award-Winning Business Applications*. John Wiley & Sons. ISBN 0-471-14717-6.

- Jacobson, Ivar (1992). *Object-Oriented Software Engineering: A Use Case-Driven Approach*. Addison-Wesley. ISBN 0-201-54435-0.
- Kay, Alan. *The Early History of Smalltalk* (<http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html>).
- Meyer, Bertrand (1997). *Object-Oriented Software Construction*. Prentice Hall. ISBN 0-13-629155-4.
- Rumbaugh, James; Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen (1991). *Object-Oriented Modeling and Design*. Prentice Hall. ISBN 0-13-629841-9.
- Taylor, David A. (1992). *Object-Oriented Information Systems - Planning and Implementation*. John Wiley & Sons. ISBN 0-471-54364-0.
- Schreiner, Axel-Tobias (1993). *Object oriented programming with ANSI-C*. Hanser. ISBN 3-446-17426-5. hdl:1850/8544.

External links

- Object-oriented programming (<http://www.dmoz.org/Computers/Programming/Methodologies/Object-Oriented/>) at the Open Directory Project
- Chapter on implementing OOP in the programming language C (<http://www.polberger.se/components/read/demystifying-dynamic-dispatch-wikipedia.html>) by David Polberger
- Discussion about the flaws of OOD (<http://software.intel.com/en-us/blogs/2008/08/22/flaws-of-object-oriented-modeling/>)
- OOP Concepts (Java Tutorials) (<http://java.sun.com/docs/books/tutorial/java/concepts/index.html>)
- Intel blog - killing OOP (<http://software.intel.com/en-us/blogs/2011/05/18/the-first-thing-we-do-lets-kill-all-the-object-oriented-programming/>)

Smalltalk

Smalltalk

<i>Smalltalk-80: The Language and its Implementation</i> , a.k.a. the "Blue book", a seminal book on the language	
Paradigm(s)	object-oriented
Appeared in	1972 (development began in 1969)
Designed by	Alan Kay, Dan Ingalls, Adele Goldberg
Developer	Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Diana Merry, Scott Wallace, Peter Deutsch and Xerox PARC
Stable release	Smalltalk-80 version 2 (1980)
Typing discipline	strong, dynamic
Major implementations	Pharo, Squeak, GNU Smalltalk, VisualWorks, Dolphin Smalltalk, Smalltalk/X, VA Smalltalk
Influenced by	Lisp, Simula, Logo, Sketchpad
Influenced	Objective-C, Self, Java, PHP 5, Logtalk, Dylan, AppleScript, Lisaac, NewtonScript, Python, Ruby, Groovy, Scala, Perl 6, Common Lisp Object System, Falcon, Io, Ioke, Fancy, Dart
OS	Cross-platform (multi-platform)
 Smalltalk at Wikibooks	

Smalltalk is an object-oriented, dynamically typed, reflective programming language. Smalltalk was created as the language to underpin the "new world" of computing exemplified by "human–computer symbiosis."^[1] It was designed and created in part for educational use, more so for constructionist learning, at the Learning Research Group (LRG) of Xerox PARC by Alan Kay, Dan Ingalls, Adele Goldberg, Ted Kaehler, Scott Wallace, and others during the 1970s.

The language was first generally released as Smalltalk-80. Smalltalk-like languages are in continuing active development, and have gathered loyal communities of users around them. ANSI Smalltalk was ratified in 1998 and represents the standard version of Smalltalk.^[2]

History

There are a large number of Smalltalk variants.^[3] The unqualified word *Smalltalk* is often used to indicate the Smalltalk-80 language, the first version to be made publicly available and created in 1980.

Smalltalk was the product of research led by Alan Kay at Xerox Palo Alto Research Center (PARC); Alan Kay designed most of the early Smalltalk versions, which Dan Ingalls implemented. The first version, known as Smalltalk-71, was created by Ingalls in a few mornings on a bet that a programming language based on the idea of message passing inspired by Simula could be implemented in "a page of code."^[1] A later variant actually used for research work is now known as Smalltalk-72 and influenced the development of the Actor model. Its syntax and execution model were very different from modern Smalltalk variants.

After significant revisions which froze some aspects of execution semantics to gain performance (by adopting a Simula-like class inheritance model of execution), Smalltalk-76 was created. This system had a development environment featuring most of the now familiar tools, including a class library code browser/editor. Smalltalk-80 added metaclasses, to help maintain the "everything is an object" (except private instance variables) paradigm by associating properties and behavior with individual classes, and even primitives such as integer and boolean values (for example, to support different ways of creating instances).

Smalltalk-80 was the first language variant made available outside of PARC, first as Smalltalk-80 Version 1, given to a small number of firms (Hewlett-Packard, Apple Computer, Tektronix, and DEC) and universities (UC Berkeley) for "peer review" and implementation on their platforms. Later (in 1983) a general availability implementation, known as Smalltalk-80 Version 2, was released as an image (platform-independent file with object definitions) and a virtual machine specification. ANSI Smalltalk has been the standard language reference since 1998.^[4]

Two of the currently popular Smalltalk implementation variants are descendants of those original Smalltalk-80 images. Squeak is an open source implementation derived from Smalltalk-80 Version 1 by way of Apple Smalltalk. VisualWorks is derived from Smalltalk-80 version 2 by way of Smalltalk-80 2.5 and ObjectWorks (both products of ParcPlace Systems, a Xerox PARC spin-off company formed to bring Smalltalk to the market). As an interesting link between generations, in 2002 Vassili Bykov implemented Hobbes, a virtual machine running Smalltalk-80 inside VisualWorks.^[5] (Dan Ingalls later ported Hobbes to Squeak.)

During the late 1980s to mid-1990s, Smalltalk environments—including support, training and add-ons—were sold by two competing organizations: ParcPlace Systems and Digitalk, both California based. ParcPlace Systems tended to focus on the Unix/Sun Microsystems market, while Digitalk focused on Intel-based PCs running Microsoft Windows or IBM's OS/2. Both firms struggled to take Smalltalk mainstream due to Smalltalk's substantial memory needs, limited run-time performance, and initial lack of supported connectivity to SQL-based relational database servers. While the high price of ParcPlace Smalltalk limited its market penetration to mid-sized and large commercial organizations, the Digitalk products initially tried to reach a wider audience with a lower price. IBM initially supported the Digitalk product, but then entered the market with a Smalltalk product in 1995 called VisualAge/Smalltalk. Easel introduced Enfin at this time on Windows and OS/2. Enfin became far more popular in Europe, as IBM introduced it into IT shops before their development of IBM Smalltalk (later VisualAge). Enfin was later acquired by Cincom Systems, and is now sold under the name ObjectStudio, and is part of the Cincom Smalltalk product suite.

In 1995, ParcPlace and Digitalk merged into ParcPlace-Digitalk and then rebranded in 1997 as ObjectShare, located in Irvine, CA. ObjectShare (NASDAQ: OBJS) was traded publicly until 1999, when it was delisted and dissolved. The merged firm never managed to find an effective response to Java as to market positioning, and by 1997 its owners were looking to sell the business. In 1999, Seagull Software acquired the ObjectShare Java development lab (including the original Smalltalk/V and Visual Smalltalk development team), and still owns VisualSmalltalk, although worldwide distribution rights for the Smalltalk product remained with ObjectShare who then sold them to Cincom.^[6] VisualWorks was sold to Cincom and is now part of Cincom Smalltalk. Cincom has backed Smalltalk strongly, releasing multiple new versions of VisualWorks and ObjectStudio each year since 1999.

Cincom, Gemstone and Object Arts, plus other vendors continue to sell Smalltalk environments. IBM has 'end of life'd VisualAge Smalltalk having in the late 1990s decided to back Java and it is, as of 2006, supported by Instantiations, Inc.^[7] which has renamed the product VA Smalltalk and released several new versions. The open Squeak implementation has an active community of developers, including many of the original Smalltalk community, and has recently been used to provide the Etoys environment on the OLPC project, a toolkit for developing collaborative applications Croquet Project, and the Open Cobalt virtual world application. GNU Smalltalk is a free software implementation of a derivative of Smalltalk-80 from the GNU project. Last but not least Pharo Smalltalk (a fork of Squeak oriented towards research and use in commercial environments) a new and clean MIT licensed open source Smalltalk that brings fresh ideas and interest into the Smalltalk market and scene.

A significant development, that has spread across all current Smalltalk environments, is the increasing usage of two web frameworks, Seaside and AIDA/Web, to simplify the building of complex web applications. Seaside has seen considerable market interest with Cincom, Gemstone and Instantiations incorporating and extending it.

Influences

John Shoch, a member of the LRG at PARC, acknowledged in his 1979 paper Smalltalk's debt to Plato's theory of forms in which an ideal archetype becomes the template from which other objects are derived.^[8] Smalltalk has influenced the wider world of computer programming in four main areas. It inspired the syntax and semantics of other computer programming languages. Secondly, it was a prototype for a model of computation known as message passing. Thirdly, its WIMP GUI inspired the windowing environments of personal computers in the late twentieth and early twenty-first centuries, so much so that the windows of the first Macintosh desktop look almost identical to the MVC windows of Smalltalk-80. Finally, the integrated development environment was the model for a generation of visual programming tools that look like Smalltalk's code browsers and debuggers.

Python and Ruby have reimplemented some Smalltalk ideas in an environment similar to that of AWK or Perl. The Smalltalk "metamodel" also serves as the inspiration for the object model design of Perl 6.

The syntax and runtime behaviour of the Objective-C programming language is strongly influenced by Smalltalk.

There is also a modular Smalltalk-like implementation designed for scripting called S#, or Script.NET. S# uses just-in-time compilation technology and supports an extended Smalltalk-like language written by David Simmons of Smallscript Corp.^{[9][10]}

Several programming languages like Self, ECMAScript/JavaScript, and Newspeak have taken the ideas of Smalltalk in new directions.

Object-oriented programming

As in other object-oriented languages, the central concept in Smalltalk-80 (but not in Smalltalk-72) is that of an *object*. An object is always an *instance* of a *class*. Classes are "blueprints" that describe the properties and behavior of their instances. For example, a GUI's window class might declare that windows have properties such as the label, the position and whether the window is visible or not. The class might also declare that instances support operations such as opening, closing, moving and hiding. Each particular window object would have its own values of those properties, and each of them would be able to perform operations defined by its class.

A Smalltalk object can do exactly three things:

1. Hold state (references to other objects).
2. Receive a message from itself or another object.
3. In the course of processing a message, send messages to itself or another object.

The state an object holds is always private to that object. Other objects can query or change that state only by sending requests (messages) to the object to do so. Any message can be sent to any object: when a message is received, the receiver determines whether that message is appropriate. Alan Kay has commented that despite the attention given to objects, messaging is the most important concept in Smalltalk: "The big idea is 'messaging' -- that is what the kernel of Smalltalk/Squeak is all about (and it's something that was never quite completed in our Xerox PARC phase)."^[11]

Smalltalk is a "pure" object-oriented programming language, meaning that, unlike Java and C++, there is no difference between values which are objects and values which are primitive types. In Smalltalk, primitive values such as integers, booleans and characters are also objects, in the sense that they are instances of corresponding classes, and operations on them are invoked by sending messages. A programmer can change the classes that implement primitive values, so that new behavior can be defined for their instances—for example, to implement new control structures—or even so that their existing behavior will be changed. This fact is summarized in the commonly heard phrase "In Smalltalk everything is an object", which may be more accurately expressed as "all values are objects", as variables are not.

Since all values are objects, classes themselves are also objects. Each class is an instance of the *metaclass* of that class. Metaclasses in turn are also objects, and are all instances of a class called Metaclass. Code blocks are also objects.

Reflection

Smalltalk-80 is a totally reflective system, implemented in Smalltalk-80 itself. Smalltalk-80 provides both structural and computational reflection. Smalltalk is a structurally reflective system whose structure is defined by Smalltalk-80 objects. The classes and methods that define the system are themselves objects and fully part of the system that they help define. The Smalltalk compiler compiles textual source code into method objects, typically instances of CompiledMethod. These get added to classes by storing them in a class's method dictionary. The part of the class hierarchy that defines classes can add new classes to the system. The system is extended by running Smalltalk-80 code that creates or defines classes and methods. In this way a Smalltalk-80 system is a "living" system, carrying around the ability to extend itself at run time.

Since the classes are themselves objects, they can be asked questions such as "what methods do you implement?" or "what fields/slots/instance variables do you define?". So objects can easily be inspected, copied, (de)serialized and so on with generic code that applies to any object in the system.

Smalltalk-80 also provides computational reflection, the ability to observe the computational state of the system. In languages derived from the original Smalltalk-80 the current activation of a method is accessible as an object named via a pseudo-variable (one of the six reserved words), `thisContext`. By sending messages to `thisContext` a method activation can ask questions like "who sent this message to me". These facilities make it possible to implement co-routines or Prolog-like back-tracking without modifying the virtual machine. The exception system is implemented using this facility. One of the more interesting uses of this is in the Seaside web framework which relieves the programmer of dealing with the complexity of a Web Browser's back button by storing continuations for each edited page and switching between them as the user navigates a web site. Programming the web server using Seaside can then be done using a more conventional programming style.

When an object is sent a message that it does not implement, the virtual machine sends the object the `doesNotUnderstand:` message with a reification of the message as an argument. The message (another object, an instance of `Message`) contains the selector of the message and an `Array` of its arguments. In an interactive Smalltalk system the default implementation of `doesNotUnderstand:` is one that opens an error window (a `Notifier`) reporting the error to the user. Through this and the reflective facilities the user can examine the context in which the error occurred, redefine the offending code, and continue, all within the system, using Smalltalk-80's reflective facilities.

Another important use of `doesNotUnderstand:` is *intercession*. One can create a class that does not define any methods other than `doesNotUnderstand:` and does not inherit from any other class. The instances of this class effectively understand no messages. So every time a message is sent to these instances they actually get sent `doesNotUnderstand:`, hence they intercede in the message sending process. Such objects are called proxies. By implementing `doesNotUnderstand:` appropriately, one can create distributed systems where proxies forward messages across a network to other Smalltalk systems (a facility common in systems like CORBA, COM+ and RMI but first pioneered in Smalltalk-80 in the 1980s), and persistent systems where changes in state are written to a database and the like. An example of this latter is Logic Arts' VOSS (Virtual Object Storage System) available for VA Smalltalk under dual open source and commercial licensing.

Syntax

Smalltalk-80 syntax is rather minimalist, based on only a handful of declarations and reserved words. In fact, only six "keywords" are reserved in Smalltalk: `true`, `false`, `nil`, `self`, `super`, and `thisContext`. These are actually called *pseudo-variables*, identifiers that follow the rules for variable identifiers but denote bindings that the programmer cannot change. The `true`, `false`, and `nil` pseudo-variables are singleton instances. `self` and `super` refer to the receiver of a message within a method activated in response to that message, but sends to `super` are looked up in the superclass of the method's defining class rather than the class of the receiver, which allows methods in subclasses to invoke methods of the same name in superclasses. `thisContext` refers to the current activation record. The only built-in language constructs are message sends, assignment, method return and literal syntax for some objects. From its origins as a language for children of all ages, standard Smalltalk syntax uses punctuation in a manner more like English than mainstream coding languages. The remainder of the language, including control structures for conditional evaluation and iteration, is implemented on top of the built-in constructs by the standard Smalltalk class library. (For performance reasons, implementations may recognize and treat as special some of those messages; however, this is only an optimization and is not hardwired into the language syntax.)

Literals

The following examples illustrate the most common objects which can be written as literal values in Smalltalk-80 methods.

Numbers. The following list illustrates some of the possibilities.

```
42  
-42  
123.45  
1.2345e2  
2r10010010  
16rA000
```

The last two entries are a binary and a hexadecimal number, respectively. The number before the 'r' is the radix or base. The base does not have to be a power of two; for example 36rSMALLTALK is a valid number equal to 80738163270632 decimal.

Characters are written by preceding them with a dollar sign:

```
$A
```

Strings are sequences of characters enclosed in single quotes:

```
'Hello, world!'
```

To include a quote in a string, escape it using a second quote:

```
'I said, ''Hello, world!'' to them.'
```

Double quotes do not need escaping, since single quotes delimit a string:

```
'I said, "Hello, world!" to them.'
```

Two equal strings (strings are equal if they contain all the same characters) can be different objects residing in different places in memory. In addition to strings, Smalltalk has a class of character sequence objects called `Symbol`. `Symbols` are guaranteed to be unique—there can be no two equal symbols which are different objects. Because of that, symbols are very cheap to compare and are often used for language artifacts such as message selectors (see

below).

Symbols are written as # followed by a string literal. For example:

```
# 'foo'
```

If the sequence does not include whitespace or punctuation characters, this can also be written as:

```
#foo
```

Arrays:

```
# (1 2 3 4)
```

defines an array of four integers.

Many implementations support the following literal syntax for ByteArrays:

```
# [1 2 3 4]
```

defines a ByteArray of four integers.

And last but not least, blocks (anonymous function literals)

```
[... Some smalltalk code...]
```

Blocks are explained in detail further in the text.

Many Smalltalk dialects implement additional syntaxes for other objects, but the ones above are the essentials supported by all.

Variable declarations

The two kinds of variable commonly used in Smalltalk are instance variables and temporary variables. Other variables and related terminology depend on the particular implementation. For example, VisualWorks has class shared variables and namespace shared variables, while Squeak and many other implementations have class variables, pool variables and global variables.

Temporary variable declarations in Smalltalk are variables declared inside a method (see below). They are declared at the top of the method as names separated by spaces and enclosed by vertical bars. For example:

```
| index |
```

declares a temporary variable named index. Multiple variables may be declared within one set of bars:

```
| index vowels |
```

declares two variables: index and vowels.

Assignment

A variable is assigned a value via the ':=' syntax. So:

```
vowels := 'aeiou'
```

Assigns the string 'aeiou' to the previously declared vowels variable. The string is an object (a sequence of characters between single quotes is the syntax for literal strings), created by the compiler at compile time.

In the original Parc Place image, the glyph of the underscore character (_) appeared as a left-facing arrow. Smalltalk originally accepted this left-arrow as the only assignment operator. Some modern code still contains what appear to be underscores acting as assignments, hearkening back to this original usage. Most modern Smalltalk implementations accept either the underscore or the colon-equals syntax.

Messages

The message is the most fundamental language construct in Smalltalk. Even control structures are implemented as message sends. Smalltalk adopts by default a synchronous, single dynamic message dispatch strategy (as contrasted to a synchronous, multiple dispatch strategy adopted by some other object-oriented languages).

The following example sends the message 'factorial' to number 42:

```
42 factorial
```

In this situation 42 is called the message *receiver*, while 'factorial' is the message *selector*. The receiver responds to the message by returning a value (presumably in this case the factorial of 42). Among other things, the result of the message can be assigned to a variable:

```
aRatherBigNumber := 42 factorial
```

"factorial" above is what is called a *unary message* because only one object, the receiver, is involved. Messages can carry additional objects as *arguments*, as follows:

```
2 raisedTo: 4
```

In this expression two objects are involved: 2 as the receiver and 4 as the message argument. The message result, or in Smalltalk parlance, *the answer* is supposed to be 16. Such messages are called *keyword messages*. A message can have more arguments, using the following syntax:

```
'hello world' indexOf: $o startingAt: 6
```

which answers the index of character 'o' in the receiver string, starting the search from index 6. The selector of this message is "indexOf:startingAt:", consisting of two pieces, or *keywords*.

Such interleaving of keywords and arguments is meant to improve readability of code, since arguments are explained by their preceding keywords. For example, an expression to create a rectangle using a C++ or Java-like syntax might be written as:

```
new Rectangle(100, 200);
```

It's unclear which argument is which. By contrast, in Smalltalk, this code would be written as:

```
Rectangle width: 100 height: 200
```

The receiver in this case is "Rectangle", a class, and the answer will be a new instance of the class with the specified width and height.

Finally, most of the special (non-alphabetic) characters can be used as what are called *binary messages*. These allow mathematical and logical operators to be written in their traditional form:

```
3 + 4
```

which sends the message "+" to the receiver 3 with 4 passed as the argument (the answer of which will be 7). Similarly,

```
3 > 4
```

is the message ">" sent to 3 with argument 4 (the answer of which will be false).

Notice, that the Smalltalk-80 language itself does not imply the meaning of those operators. The outcome of the above is only defined by how the receiver of the message (in this case a Number instance) responds to messages "+" and ">".

A side effect of this mechanism is operator overloading. A message ">" can also be understood by other objects, allowing the use of expressions of the form "a > b" to compare them.

Expressions

An expression can include multiple message sends. In this case expressions are parsed according to a simple order of precedence. Unary messages have the highest precedence, followed by binary messages, followed by keyword messages. For example:

```
3 factorial + 4 factorial between: 10 and: 100
```

is evaluated as follows:

1. 3 receives the message "factorial" and answers 6
2. 4 receives the message "factorial" and answers 24
3. 6 receives the message "+" with 24 as the argument and answers 30
4. 30 receives the message "between:and:" with 10 and 100 as arguments and answers true

The answer of the last message sent is the result of the entire expression.

Parentheses can alter the order of evaluation when needed. For example,

```
(3 factorial + 4) factorial between: 10 and: 100
```

will change the meaning so that the expression first computes "3 factorial + 4" yielding 10. That 10 then receives the second "factorial" message, yielding 3628800. 3628800 then receives "between:and:", answering false.

Note that because the meaning of binary messages is not hardwired into Smalltalk-80 syntax, all of them are considered to have equal precedence and are evaluated simply from left to right. Because of this, the meaning of Smalltalk expressions using binary messages can be different from their "traditional" interpretation:

```
3 + 4 * 5
```

is evaluated as "(3 + 4) * 5", producing 35. To obtain the expected answer of 23, parentheses must be used to explicitly define the order of operations:

```
3 + (4 * 5)
```

Unary messages can be *chained* by writing them one after another:

```
3 factorial factorial log
```

which sends "factorial" to 3, then "factorial" to the result (6), then "log" to the result (720), producing the result 2.85733.

A series of expressions can be written as in the following (hypothetical) example, each separated by a period. This example first creates a new instance of class Window, stores it in a variable, and then sends two messages to it.

```
| window |
window := Window new.
window label: 'Hello'.
window open
```

If a series of messages are sent to the same receiver as in the example above, they can also be written as a *cascade* with individual messages separated by semicolons:

```
Window new
label: 'Hello';
open
```

This rewrite of the earlier example as a single expression avoids the need to store the new window in a temporary variable. According to the usual precedence rules, the unary message "new" is sent first, and then "label:" and "open" are sent to the answer of "new".

Code blocks

A block of code (an anonymous function) can be expressed as a literal value (which is an object, since all values are objects.) This is achieved with square brackets:

```
[ :params | <message-expressions> ]
```

Where *:params* is the list of parameters the code can take. This means that the Smalltalk code:

```
[ :x | x + 1]
```

can be understood as:

$$f: f(x) = x + 1$$

or expressed in lambda terms as:

$$\lambda x: x + 1$$

and

```
[ :x | x + 1] value: 3
```

can be evaluated as

$$f(3) = 3 + 1$$

Or in lambda terms as:

$$(\lambda x: x + 1)3 \beta \rightarrow 4$$

The resulting block object can form a closure: it can access the variables of its enclosing lexical scopes at any time. Blocks are first-class objects.

Blocks can be executed by sending them the *value* message (compound variations exist in order to provide parameters to the block e.g. 'value:value:' and 'valueWithArguments:').

The literal representation of blocks was an innovation which on the one hand allowed certain code to be significantly more readable; it allowed algorithms involving iteration to be coded in a clear and concise way. Code that would typically be written with loops in some languages can be written concisely in Smalltalk using blocks, sometimes in a single line. But more importantly blocks allow control structure to be expressed using messages and polymorphism, since blocks defer computation and polymorphism can be used to select alternatives. So if-then-else in Smalltalk is written and implemented as

```
expr ifTrue: [statements to evaluate if expr] ifFalse: [statements to evaluate if not expr]
```

True methods for evaluation

ifTrue: trueAlternativeBlock **ifFalse:** falseAlternativeBlock

```
^trueAlternativeBlock value
```

False methods for evaluation

ifTrue: trueAlternativeBlock **ifFalse:** falseAlternativeBlock

```
^falseAlternativeBlock value
```

```
positiveAmounts := allAmounts select: [:anAmount | anAmount isPositive]
```

Note that this is related to functional programming, wherein patterns of computation (here selection) are abstracted into higher-order functions. For example, the message *select:* on a Collection is equivalent to the higher-order

function filter on an appropriate functor.

Control structures

Control structures do not have special syntax in Smalltalk. They are instead implemented as messages sent to objects. For example, conditional execution is implemented by sending the message `ifTrue:` to a Boolean object, passing as an argument the block of code to be executed if and only if the Boolean receiver is true.

The following code demonstrates this:

```
result := a > b
    ifTrue: [ 'greater' ]
    ifFalse: [ 'less or equal' ]
```

Blocks are also used to implement user-defined control structures, enumerators, visitors, pluggable behavior and many other patterns. For example:

```
| aString vowels |
aString := 'This is a string'.
vowels := aString select: [:aCharacter | aCharacter isVowel].
```

In the last line, the string is sent the message `select:` with an argument that is a code block literal. The code block literal will be used as a predicate function that should answer true if and only if an element of the String should be included in the Collection of characters that satisfy the test represented by the code block that is the argument to the "select:" message.

A String object responds to the "select:" message by iterating through its members (by sending itself the message "do:"), evaluating the selection block ("aBlock") once with each character it contains as the argument. When evaluated (by being sent the message "value: each"), the selection block (referenced by the parameter "aBlock", and defined by the block literal "[:aCharacter | aCharacter isVowel]"), answers a boolean, which is then sent "ifTrue:". If the boolean is the object true, the character is added to a string to be returned. Because the "select:" method is defined in the abstract class Collection, it can also be used like this:

```
| rectangles aPoint collisions |
rectangles := OrderedCollection
    with: (Rectangle left: 0 right: 10 top: 100 bottom: 200)
    with: (Rectangle left: 10 right: 10 top: 110 bottom: 210).
aPoint := Point x: 20 y: 20.
collisions := rectangles select: [:aRect | aRect containsPoint: aPoint].
```

Classes

This is a stock class definition:

```
Object subclass: #MessagePublisher
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Smalltalk Examples'
```

Often, most of this definition will be filled in by the environment. Notice that this is actually a message to the "Object"-class to create a subclass called "MessagePublisher". In other words: classes are first-class objects in Smalltalk which can receive messages just like any other object and can be created dynamically at execution time.

Methods

When an object receives a message, a method matching the message name is invoked. The following code defines a method `publish`, and so defines what will happen when this object receives the '`publish`' message.

```
publish
  Transcript show: 'Hello, World!'
```

The following method demonstrates receiving multiple arguments and returning a value:

```
quadMultiply: i1 and: i2
  "This method multiplies the given numbers by each other and the
  result by 4."
  | mul |
  mul := i1 * i2.
  ^mul * 4
```

The method's name is `#quadMultiply:and:.`. The return value is specified with the `^` operator.

Note that objects are responsible for determining dynamically at runtime which method to execute in response to a message—while in many languages this may be (sometimes, or even always) determined statically at compile time.

Instantiating classes

The following code:

```
MessagePublisher new
```

creates (and returns) a new instance of the `MessagePublisher` class. This is typically assigned to a variable:

```
publisher := MessagePublisher new
```

However, it is also possible to send a message to a temporary, anonymous object:

```
MessagePublisher new publish
```

Hello World example

In the following code, the message "`show:`" is sent to the object "`Transcript`" with the String literal '`Hello, world!`' as its argument. Invocation of the "`show:`" method causes the characters of its argument (the String literal '`Hello, world!`') to be displayed in the transcript ("terminal") window.

```
Transcript show: 'Hello, world!'.
```

Note that a Transcript window would need to be open in order to see the results of this example.

Image-based persistence

Most popular programming systems separate static program code (in the form of class definitions, functions or procedures) from dynamic, or run time, program state (such as objects or other forms of program data). They load program code when a program starts, and any prior program state must be recreated explicitly from configuration files or other data sources. Any settings the program (and programmer) does not explicitly save must be set up again for each restart. A traditional program also loses much useful document information each time a program saves a file, quits, and reloads. This loses details such as undo history or cursor position. Image based systems don't force losing all that just because a computer is turned off, or an OS updates.

Many Smalltalk systems, however, do not differentiate between program data (objects) and code (classes). In fact, classes are objects themselves. Therefore most Smalltalk systems store the entire program state (including both Class and non-Class objects) in an image file. The image can then be loaded by the Smalltalk virtual machine to restore a Smalltalk-like system to a prior state. This was inspired by FLEX,^[12] a language created by Alan Kay and described in his M.Sc. thesis.

Other languages that model application code as a form of data, such as Lisp, often use image-based persistence as well.

Smalltalk images are similar to (restartable) core dumps and can provide the same functionality as core dumps, such as delayed or remote debugging with full access to the program state at the time of error.

Level of access

Everything in Smalltalk-80 is available for modification from within a running program. This means that, for example, the IDE can be changed in a running system without restarting it. In some implementations, the syntax of the language or the garbage collection implementation can also be changed on the fly. Even the statement `true` become: `false` is valid in Smalltalk, although executing it is not recommended. When used judiciously, this level of flexibility allows for one of the shortest required times for new code to enter a production system.

Just-in-time compilation

Smalltalk programs are usually compiled to bytecode, which is then interpreted by a virtual machine or dynamically translated into machine-native code.

List of implementations

- Amber Smalltalk Smalltalk running atop JavaScript
- Athena, Smalltalk scripting engine for Java >=1.6
- Bistro
- Cincom has the following Smalltalk products: ObjectStudio, VisualWorks and WebVelocity.
 - Visual Smalltalk Enterprise, and family, including Smalltalk/V
- Exept Software, Smalltalk/X
- F-Script
- Gemstone, GemStone/S
- GNU Smalltalk
 - Étoilé Pragmatic Smalltalk, Smalltalk for Étoilé, a GNUstep-based user environment
 - StepTalk, GNUstep scripting framework uses Smalltalk language on an Objective-C runtime
- Instantiations, VA Smalltalk being the follow-on to IBM VisualAge Smalltalk
 - VisualAge Smalltalk
- Little Smalltalk
- Object Arts, Dolphin Smalltalk
- Object Connect, Smalltalk MT Smalltalk for Windows
 - LSW Vision-Smalltalk have partnered with Object Arts
- Panda Smalltalk, open source engine, written in C, has no dependencies except libc
- Pharo Smalltalk, Pharo Project's open-source multi-platform Smalltalk
- Pocket Smalltalk, runs on Palm Pilot
- Refactory, produces #Smalltalk
- Smalltalk YX

- Smalltalk/X
- Squeak, open source Smalltalk
 - Cog, JIT VM written in Squeak Smalltalk
 - CogDroid, port of non-JIT variant of Cog VM to Android
 - eToys, eToys visual programming system for learning
 - iSqueak, Squeak interpreter port for iOS devices, iPhone/iPad
 - JSqueak, Squeak interpreter written in Java
 - Potato, Squeak interpreter written in Java, a direct derivative of JSqueak
 - RoarVM, RoarVM is a multi- and manycore interpreter for Squeak and Pharo
- Strongtalk, for Windows, offers optional strong typing
- Susie, a light-weight scripting engine using Smalltalk as the language, based on Public Domain SmallTalk.

References

- [1] Kay, Alan. "The Early History of Smalltalk" (<http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html>). . Retrieved 2007-09-13.
- [2] <http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html>
- [3] "Versions" (<http://www.smalltalk.org/versions>). Smalltalk.org. . Retrieved 2007-09-13.
- [4] "ANSI Smalltalk Standard" (<http://www.smalltalk.org/versions/ANSIStandardSmalltalk.html>). Smalltalk.org. . Retrieved 2007-09-13.
- [5] Hobbes (<http://wiki.cs.uiuc.edu/VisualWorks/Smalltalk-80+in+a+box>)
- [6] "History" (<http://www.seagullsoftware.com/about/history.html#1999>). Seagull Software. . Retrieved 2007-09-13.
- [7] VisualAge Smalltalk Transition FAQ (<http://www.instantiations.com/company/ibm-transition.html>)
- [8] "Smalltalk, Objects, and Design", Chamond Liu, p. 30, iUniverse reprint, 2000, ISBN 1-58348-490-6,
- [9] S# (<http://www.ssharp.org>)
- [10] Smallscript Corp. (<http://www.smallscript.com>)
- [11] Kay, Alan (October 10, 1998). "Prototypes vs Classes (e-mail on Squeak list)" (<http://lists.squeakfoundation.org/pipermail/squeak-dev/1998-October/017019.html>). .
- [12] FLEX: A flexible extendable language (<http://www.mprove.de/diplom/gui/kay68.html>)

Further reading

- Goldberg, Adele (December 1983). *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley. ISBN 0-201-11372-4.
- Adele Goldberg & Alan Kay, ed. (March 1976). *Smalltalk-72 Instruction Manual* (http://www.bitsavers.org/pdf/xerox/parc/techReports/Smalltalk-72_Instruction_Manual_Mar76.pdf). Palo Alto, California: Xerox Palo Alto Research Center. Retrieved 2011-11-11.
- Goldberg, Adele; Robson, David (May 1983). *Smalltalk-80: The Language and its Implementation* (<http://stephane.ducasse.free.fr/FreeBooks/BlueBook/Bluebook.pdf>). Addison-Wesley. ISBN 0-201-11371-6.
- Goldberg, Adele; Robson, David (11 January 1989). *Smalltalk 80: The Language*. Addison-Wesley. ISBN 0-201-13688-0.
- Kay, Alan C. (March 1993). "The Early History of Smalltalk" (<http://www.metaobject.com/papers/Smallhistory.pdf>). *ACM SIGPLAN Notices* (ACM) **28** (3): 69–95. doi:10.1145/155360.155364.
- Glen Krasner, ed. (August 1983). *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley. ISBN 0-201-11669-3.
- Nierstrasz, Oscar; Ducasse, Stéphane; Pollet, Damien; Black, Andrew P. (2009-10-07). *Squeak by Example* (<http://www.squeakbyexample.org>). Kehrsatz, Switzerland: Square Bracket Associates. ISBN 3-9523341-0-3.
- Nierstrasz, Oscar; Ducasse, Stéphane; Pollet, Damien; Black, Andrew P. (February 23, 2010). *Pharo by Example* (<http://pharobyexample.org/>). Kehrsatz, Switzerland: Square Bracket Publishing. ISBN 978-3-9523341-4-0.
- Winston, Patrick Henry (September 3, 1997). *On to Smalltalk*. MIT, USA: Addison Wesley. ISBN 978-0201498271.

- "Special Issue on Smalltalk" (http://www.em.net/Byte_1981_08/Byte_1981_08_150dpi.pdf). *BYTE* (McGraw-Hill) **6** (8). August 1981. Retrieved 2011-11-11.
- Ingalls, Dan (August 1981). "Design Principles Behind Smalltalk" (http://web.archive.org/web/20060616024135/www.ipa.net/~dwighth/smalltalk/byte_aug81/design_principles_behind_smalltalk.html). *BYTE* (McGraw-Hill) **6** (8). Retrieved 2011-11-11.

External links

- The World of Smalltalk (<http://www.world.st/>), Smalltalk books and videos, implementations, frameworks and tools, blogs and mailing lists
- Planet Smalltalk (<http://planet.smalltalk.org/>), an aggregator of Smalltalk blog posts
- Downloadable books on Smalltalk (<http://stephane.ducasse.free.fr/FreeBooks.html>), permission obtained to make these books freely available, over 25 full texts scanned from print
- Smalltalk (<http://www.dmoz.org/Computers/Programming/Languages/Smalltalk/>) at the Open Directory Project
- ESUG (European Smalltalk Users Group) (<http://www.esug.org>), non-profit organization with commercial and academic members, has various promotion activities including a yearly event since 1993
- STIC (Smalltalk Industry Council) (<http://stic.st>), promoting Smalltalk on behalf of the Smalltalk community
- La Fundacion Argentina de Smalltalk (FAST) (<http://www.fast.org.ar>), Organizer of annual Smalltalk conference in Argentina
- ClubSmalltalk (<http://www.clubsmalltalk.org/>), a Latin American group with a website in English to promote the Smalltalk technology
- Smalltalk.org (<http://www.smalltalk.org/>), advocacy site

Common Lisp

Common Lisp

Paradigm(s)	Multi-paradigm: procedural, functional, object-oriented, meta, reflective, generic
Appeared in	1984, 1994 for ANSI Common Lisp
Developer	ANSI X3J13 committee
Typing discipline	dynamic, strong
Scope	lexical, optionally dynamic
Major implementations	Allegro CL, ABCL, CLISP, Clozure CL, CMUCL, Corman Common Lisp, ECL, GCL, LispWorks, MKCL, Movitz, Scieneer CL, SBCL, Symbolics Common Lisp
Dialects	CLtL1, CLtL2, ANSI Common Lisp
Influenced by	Lisp, Lisp Machine Lisp, MacLisp, Scheme, InterLisp
Influenced	Clojure, Dylan, Emacs Lisp, EuLisp, Java, ISLISP, Le Lisp, SKILL, SubL
OS	Cross-platform
Website	common-lisp.net ^[1]
Family	Lisp

Common Lisp (CL) is a dialect of the Lisp programming language, published in ANSI standard document *ANSI INCITS 226-1994 (R2004)*, (formerly *X3.226-1994 (R1999)*).^[2] From the ANSI Common Lisp standard the Common Lisp HyperSpec has been derived^[3] for use with web browsers. Common Lisp was developed to standardize the divergent variants of Lisp (though mainly the MacLisp variants) which predated it, thus it is not an implementation but rather a language specification. Several implementations of the Common Lisp standard are available, including free and open source software and proprietary products.

Common Lisp is a general-purpose, multi-paradigm programming language. It supports a combination of procedural, functional, and object-oriented programming paradigms. As a dynamic programming language, it facilitates evolutionary and incremental software development, with iterative compilation into efficient run-time programs.

It also supports optional type annotation and casting, which can be added as necessary at the later profiling and optimization stages, to permit the compiler to generate more efficient code. For instance, `fixnum` can hold an unboxed integer in a range supported by the hardware and implementation, permitting more efficient arithmetic than on big integers or arbitrary precision types. Similarly, the compiler can be told on a per-module or per-function basis which type safety level is wanted, using `optimize` declarations.

Common Lisp includes CLOS, an object system that supports multimethods and method combinations. It is extensible through standard features such as Lisp macros (compile-time code rearrangement accomplished by the program itself) and reader macros (extension of syntax to give special meaning to characters reserved for users for this purpose).

Though Common Lisp is not as popular as some non-Lisp languages, many of its features have made their way into other, more widely used programming languages and systems (see Greenspun's Tenth Rule).

Syntax

Common Lisp is a dialect of Lisp; it uses S-expressions to denote both code and data structure. Function and macro calls are written as lists, with the name of the function first, as in these examples:

```
(+ 2 2) ; adds 2 and 2, yielding 4.

(defvar *x*) ; Ensures that a variable *x* exists,
              ; without giving it a value. The asterisks are part
of
              ; the name. The symbol *x* is also hereby endowed
with
              ; the property that subsequent bindings of it are
dynamic,
              ; rather than lexical.
(setf *x* 42.1) ; sets the variable *x* to the floating-point value
42.1

;; Define a function that squares a number:
(defun square (x)
  (* x x))

;; Execute the function:
(square 3) ; Returns 9

;; the 'let' construct creates a scope for local variables. Here
;; the variable 'a' is bound to 6 and the variable 'b' is bound
;; to 4. Inside the 'let' is a 'body', where the last computed value
is returned.
;; Here the result of adding a and b is returned from the 'let'
expression.
;; The variables a and b have lexical scope, unless the symbols have
been
;; marked as special variables (for instance by a prior DEFVAR).
(let ((a 6)
      (b 4))
  (+ a b)) ; returns 10
```

Data types

Common Lisp has many data types—more than many other languages.

Scalar types

Number types include integers, ratios, floating-point numbers, and complex numbers.^[4] Common Lisp uses bignums to represent numerical values of arbitrary size and precision. The ratio type represents fractions exactly, a facility not available in many languages. Common Lisp automatically coerces numeric values among these types as appropriate.

The Common Lisp *character* type is not limited to ASCII characters. Most modern implementations allow Unicode characters.^[5]

The *symbol* type is common to Lisp languages, but largely unknown outside them. A symbol is a unique, named data object with several parts: name, value, function, property list and package. Of these, *value cell* and *function cell* are

the most important. Symbols in Lisp are often used similarly to identifiers in other languages: to hold the value of a variable; however there are many other uses. Normally, when a symbol is evaluated, its value is returned. Some symbols evaluate to themselves, for example all symbols in the keyword package are self-evaluating. Boolean values in Common Lisp are represented by the self-evaluating symbols T and NIL. Common Lisp has namespaces for symbols, called 'packages'.

A number of functions are available for rounding scalar numeric values in various ways. The function `round` rounds the argument to the nearest integer, with halfway cases rounded to the even integer. The functions `truncate`, `floor`, and `ceiling` round towards zero, down, or up respectively. All these functions return the discarded fractional part as a secondary value. For example, `(floor -2.5)` yields -3, 0.5; `(ceiling -2.5)` yields -2, -0.5; `(round 2.5)` yields 2, 0.5; and `(round 3.5)` yields 4, -0.5.

Data structures

Sequence types in Common Lisp include lists, vectors, bit-vectors, and strings. There are many operations which can work on any sequence type.

As in almost all other Lisp dialects, *lists* in Common Lisp are composed of *conses*, sometimes called *cons cells* or *pairs*. A cons is a data structure with two slots, called its *car* and *cdr*. A list is a linked chain of conses. Each cons's car refers to a member of the list (possibly another list). Each cons's cdr refers to the next cons—except for the last cons, whose cdr refers to the `nil` value. Conses can also easily be used to implement trees and other complex data structures; though it is usually advised to use structure or class instances instead. It is also possible to create circular data structures with conses.

Common Lisp supports multidimensional *arrays*, and can dynamically resize arrays if required. Multidimensional arrays can be used for matrix mathematics. A *vector* is a one-dimensional array. Arrays can carry any type as members (even mixed types in the same array) or can be specialized to contain a specific type of members, as in a vector of integers. Many implementations can optimize array functions when the array used is type-specialized. Two type-specialized array types are standard: a *string* is a vector of characters, while a *bit-vector* is a vector of bits.

Hash tables store associations between data objects. Any object may be used as key or value. Hash tables, like arrays, are automatically resized as needed.

Packages are collections of symbols, used chiefly to separate the parts of a program into namespaces. A package may *export* some symbols, marking them as part of a public interface. Packages can use other packages.

Structures, similar in use to C structs and Pascal records, represent arbitrary complex data structures with any number and type of fields (called *slots*). Structures allow single-inheritance.

Classes are similar to structures, but offer more dynamic features and multiple-inheritance. (See CLOS). Classes have been added late to Common Lisp and there is some conceptual overlap with structures. Objects created of classes are called *Instances*. A special case are Generic Functions. Generic Functions are both functions and instances.

Functions

Common Lisp supports first-class functions. For instance, it is possible to write functions that take other functions as arguments or return functions as well. This makes it possible to describe very general operations.

The Common Lisp library relies heavily on such higher-order functions. For example, the `sort` function takes a relational operator as an argument and key function as an optional keyword argument. This can be used not only to sort any type of data, but also to sort data structures according to a key.

```
;; Sorts the list using the > and < function as the relational operator.  
(sort (list 5 2 6 3 1 4) #'>)      ; Returns (6 5 4 3 2 1)  
(sort (list 5 2 6 3 1 4) #'<)      ; Returns (1 2 3 4 5 6)
```

```
;; Sorts the list according to the first element of each sub-list.
(sort (list '(9 A) '(3 B) '(4 C)) #'< :key #'first) ; Returns ((3 B) (4 C) (9 A))
```

The evaluation model for functions is very simple. When the evaluator encounters a form (*F A₁ A₂ ...*) then it is to assume that the symbol named *F* is one of the following:

1. A special operator (easily checked against a fixed list)
2. A macro operator (must have been defined previously)
3. The name of a function (default), which may either be a symbol, or a sub-form beginning with the symbol *lambda*.

If *F* is the name of a function, then the arguments *A₁, A₂, ..., An* are evaluated in left-to-right order, and the function is found and invoked with those values supplied as parameters.

Defining functions

The macro `defun` defines functions where a function definition gives the name of the function, the names of any arguments, and a function body:

```
(defun square (x)
  (* x x))
```

Function definitions may include *declarations*, which provide hints to the compiler about optimization settings or the data types of arguments. They may also include *documentation strings* (docstrings), which the Lisp system may use to provide interactive documentation:

```
(defun square (x)
  "Calculates the square of the single-float x."
  (declare (single-float x) (optimize (speed 3) (debug 0) (safety 1)))
  (the single-float (* x x)))
```

Anonymous functions (function literals) are defined using `lambda` expressions, e.g. `(lambda (x) (* x x))` for a function that squares its argument. Lisp programming style frequently uses higher-order functions for which it is useful to provide anonymous functions as arguments.

Local functions can be defined with `flet` and `labels`.

```
(flet ((square (x)
    (* x x)))
  (square 3))
```

There are a number of other operators related to the definition and manipulation of functions. For instance, a function may be recompiled with the `compile` operator. (Some Lisp systems run functions in an interpreter by default unless instructed to compile; others compile every entered function on the fly).

Defining generic functions and methods

The macro `defgeneric` defines generic functions. The macro `defmethod` defines methods. Generic functions are a collection of methods.

Methods can specialize their parameters over classes or objects.

When a generic function is called, multiple-dispatch will determine the correct method to use.

```
(defgeneric add (a b))

(defmethod add ((a number) (b number))
  (+ a b))

(defmethod add ((a vector) (b number))
  (map 'vector (lambda (n) (+ n b)) a))

(defmethod add ((a vector) (b vector))
  (map 'vector #'+ a b))

(add 2 3)                      ; returns 5
(add #(1 2 3 4) 7)              ; returns #(8 9 10 11)
(add #(1 2 3 4) #(4 3 2 1)) ; returns #(5 5 5 5)
```

Generic Functions are also a first class data type. There are many more features to Generic Functions and Methods than described above.

The function namespace

The namespace for function names is separate from the namespace for data variables. This is a key difference between Common Lisp and Scheme. For Common Lisp, operators that define names in the function namespace include `defun`, `flet`, `labels`, `defmethod` and `defgeneric`.

To pass a function by name as an argument to another function, one must use the `function` special operator, commonly abbreviated as `#'`. The first `sort` example above refers to the function named by the symbol `>` in the function namespace, with the code `#'>`. Conversely, to call a function passed in such a way, one would use the `funcall` operator on the argument.

Scheme's evaluation model is simpler: there is only one namespace, and all positions in the form are evaluated (in any order) -- not just the arguments. Code written in one dialect is therefore sometimes confusing to programmers more experienced in the other. For instance, many Common Lisp programmers like to use descriptive variable names such as `list` or `string` which could cause problems in Scheme, as they would locally shadow function names.

Whether a separate namespace for functions is an advantage is a source of contention in the Lisp community. It is usually referred to as the *Lisp-1 vs. Lisp-2 debate*. Lisp-1 refers to Scheme's model and Lisp-2 refers to Common Lisp's model. These names were coined in a 1988 paper by Richard P. Gabriel and Kent Pitman, which extensively compares the two approaches.^[6]

Other types

Other data types in Common Lisp include:

- *Pathnames* represent files and directories in the filesystem. The Common Lisp pathname facility is more general than most operating systems' file naming conventions, making Lisp programs' access to files broadly portable across diverse systems.
- Input and output *streams* represent sources and sinks of binary or textual data, such as the terminal or open files.
- Common Lisp has a built-in pseudo-random number generator (PRNG). *Random state* objects represent reusable sources of pseudo-random numbers, allowing the user to seed the PRNG or cause it to replay a sequence.
- *Conditions* are a type used to represent errors, exceptions, and other "interesting" events to which a program may respond.
- *Classes* are first-class objects, and are themselves instances of classes called metaobject classes (metaclasses for short).
- *Readtables* are a type of object which control how Common Lisp's reader parses the text of source code. By controlling which readtable is in use when code is read in, the programmer can change or extend the language's syntax.

Scope

Like programs in many other programming languages, Common Lisp programs make use of names to refer to variables, functions, and many other kinds of entities. Named references are subject to scope.

The association between a name and the entity which the name refers to is called a binding.

Scope refers to the set of circumstances in which a name is determined to have a particular binding.

Determiners of scope

The circumstances which determine scope in Common Lisp include:

- the location of a reference within an expression. If it's the leftmost position of a compound, it refers to a special operator or a macro or function binding, otherwise to a variable binding or something else.
- the kind of expression in which the reference takes place. For instance, (GO X) means transfer control to label X, whereas (PRINT X) refers to the variable X. Both scopes of X can be active in the same region of program text, since tagbody labels are in a separate namespace from variable names. A special form or macro form has complete control over the meanings of all symbols in its syntax. For instance in (defclass x (a b)()), a class definition, the (a b) is a list of base classes, so these names are looked up in the space of class names, and x isn't a reference to an existing binding, but the name of a new class being derived from a and b. These facts emerge purely from the semantics of defclass. The only generic fact about this expression is that defclass refers to a macro binding; everything else is up to defclass.
- the location of the reference within the program text. For instance, if a reference to variable X is enclosed in a binding construct such as a LET which defines a binding for X, then the reference is in the scope created by that binding.
- for a variable reference, whether or not a variable symbol has been, locally or globally, declared special. This determines whether the reference is resolved within a lexical environment, or within a dynamic environment.
- the specific instance of the environment in which the reference is resolved. An environment is a run-time dictionary which maps symbols to bindings. Each kind of reference uses its own kind of environment. References to lexical variables are resolved in a lexical environment, et cetera. More than one environment can be associated with the same reference. For instance, thanks to recursion or the use of multiple threads, multiple activations of the same function can exist at the same time. These activations share the same program text, but each has its own lexical environment instance.

To understand what a symbol refers to, the Common Lisp programmer must know what kind of reference is being expressed, what kind of scope it uses if it is a variable reference (dynamic versus lexical scope), and also the run-time situation: in what environment is the reference resolved, where was the binding introduced into the environment, et cetera.

Kinds of environment

Global

Some environments in Lisp are globally pervasive. For instance, if a new type is defined, it is known everywhere thereafter. References to that type look it up in this global environment.

Dynamic

One type of environment in Common Lisp is the dynamic environment. Bindings established in this environment have dynamic extent, which means that a binding is established at the start of the execution of some construct, such as a `LET` block, and disappears when that construct finishes executing: its lifetime is tied to the dynamic activation and deactivation of a block. However, a dynamic binding is not just visible within that block; it is also visible to all functions invoked from that block. This type of visibility is known as indefinite scope. Bindings which exhibit dynamic extent (lifetime tied to the activation and deactivation of a block) and indefinite scope (visible to all functions which are called from that block) are said to have dynamic scope.

Common Lisp has support for dynamically scoped variables, which are also called special variables. Certain other kinds of bindings are necessarily dynamically scoped also, such as restarts and catch tags. Function bindings cannot be dynamically scoped using `FLET` (which only provides lexically scoped function bindings), but function objects (a first-level object in Common Lisp) can be assigned to dynamically scoped variables, bound using `LET` in dynamic scope, then called using `FUNCALL` or `APPLY`.

Dynamic scope is extremely useful because it adds referential clarity and discipline to global variables. Global variables are frowned upon in computer science as potential sources of error, because they can give rise to ad-hoc, covert channels of communication among modules that lead to unwanted, surprising interactions.

In Common Lisp, a special variable which has only a top-level binding behaves just like a global variable in other programming languages. A new value can be stored into it, and that value simply replaces what is in the top-level binding. Careless replacement of the value of a global variable is at the heart of bugs caused by use of global variables. However, another way to work with a special variable is to give it a new, local binding within an expression. This is sometimes referred to as "rebinding" the variable. Binding a dynamically scoped variable temporarily creates a new memory location for that variable, and associates the name with that location. While that binding is in effect, all references to that variable refer to the new binding; the previous binding is hidden. When execution of the binding expression terminates, the temporary memory location is gone, and the old binding is revealed, with the original value intact. Of course, multiple dynamic bindings for the same variable can be nested.

In Common Lisp implementations which support multithreading, dynamic scopes are specific to each thread of execution. Thus special variables serve as an abstraction for thread local storage. If one thread rebinds a special variable, this rebinding has no effect on that variable in other threads. The value stored in a binding can only be retrieved by the thread which created that binding. If each thread binds some special variable `*X*`, then `*X*` behaves like thread-local storage. Among threads which do not rebind `*X*`, it behaves like an ordinary global: all of these threads refer to the same top-level binding of `*X*`.

Dynamic variables can be used to extend the execution context with additional context information which is implicitly passed from function to function without having to appear as an extra function parameter. This is especially useful when the control transfer has to pass through layers of unrelated code, which simply cannot be extended with extra parameters to pass the additional data. A situation like this usually calls for a global variable.

That global variable must be saved and restored, so that the scheme doesn't break under recursion: dynamic variable rebinding takes care of this. And that variable must be made thread-local (or else a big mutex must be used) so the scheme doesn't break under threads: dynamic scope implementations can take care of this also.

In the Common Lisp library, there are many standard special variables. For instance, all standard I/O streams are stored in the top-level bindings of well-known special variables. The standard output stream is stored in *standard-output*.

Suppose a function foo writes to standard output:

```
(defun foo ()
  (format t "Hello, world"))
```

To capture its output in a character string, *standard-output* can be bound to a string stream and called:

```
(with-output-to-string (*standard-output*)
  (foo))
```

```
-> "Hello, world" ; gathered output returned as a string
```

Lexical

Common Lisp supports lexical environments. Formally, the bindings in a lexical environment have lexical scope and may have either indefinite extent or dynamic extent, depending on the type of namespace. Lexical scope means that visibility is physically restricted to the block in which the binding is established. References which are not textually (i.e. lexically) embedded in that block simply do not see that binding.

The tags in a TAGBODY have lexical scope. The expression (GO X) is erroneous if it is not actually embedded in a TAGBODY which contains a label X. However, the label bindings disappear when the TAGBODY terminates its execution, because they have dynamic extent. If that block of code is re-entered by the invocation of a lexical closure, it is invalid for the body of that closure to try to transfer control to a tag via GO:

```
(defvar *stashed* ; will hold a function

(tagbody
  (setf *stashed* (lambda () (go some-label)))
  (go end-label) ; skip the (print "Hello")
  some-label
  (print "Hello")
  end-label)
-> NIL
```

When the TAGBODY is executed, it first evaluates the setf form which stores a function in the special variable *stashed*. Then the (go end-label) transfers control to end-label, skipping the code (print "Hello"). Since end-label is at the end of the tagbody, the tagbody terminates, yielding NIL. Suppose that the previously remembered function is now called:

```
(funcall *stashed*) ;; Error!
```

This situation is erroneous. One implementation's response is an error condition containing the message, "GO: tagbody for tag SOME-LABEL has already been left". The function tried to evaluate (go some-label), which is lexically embedded in the tagbody, and resolves to the label. However, the tagbody isn't executing (its extent has ended), and so the control transfer cannot take place.

Local function bindings in Lisp have lexical scope, and variable bindings also have lexical scope by default. By contrast with GO labels, both of these have indefinite extent. When a lexical function or variable binding is established, that binding continues to exist for as long as references to it are possible, even after the construct which established that binding has terminated. References to a lexical variables and functions after the termination of their establishing construct are possible thanks to lexical closures.

Lexical binding is the default binding mode for Common Lisp variables. For an individual symbol, it can be switched to dynamic scope, either by a local declaration, by a global declaration. The latter may occur implicitly through the use of a construct like DEFVAR or DEFPARAMETER. It is an important convention in Common Lisp programming that special (i.e. dynamically scoped) variables have names which begin and end with an asterisk sigil * in what is called the “earmuff convention”.^[7] If adhered to, this convention effectively creates a separate namespace for special variables, so that variables intended to be lexical are not accidentally made special.

Lexical scope is useful for several reasons.

Firstly, references to variables and functions can be compiled to efficient machine code, because the run-time environment structure is relatively simple. In many cases it can be optimized to stack storage, so opening and closing lexical scopes has minimal overhead. Even in cases where full closures must be generated, access to the closure's environment is still efficient; typically each variable becomes an offset into a vector of bindings, and so a variable reference becomes a simple load or store instruction with a base-plus-offset addressing mode.

Secondly, lexical scope (combined with indefinite extent) gives rise to the lexical closure, which in turn creates a whole paradigm of programming centered around the use of functions being first-class objects, which is at the root of functional programming.

Thirdly, perhaps most importantly, even if lexical closures are not exploited, the use of lexical scope isolates program modules from unwanted interactions. Due to their restricted visibility, lexical variables are private. If one module A binds a lexical variable X, and calls another module B, references to X in B will not accidentally resolve to the X bound in A. B simply has no access to X. For situations in which disciplined interactions through a variable are desirable, Common Lisp provides special variables. Special variables allow for a module A to set up a binding for a variable X which is visible to another module B, called from A. Being able to do this is an advantage, and being able to prevent it from happening is also an advantage; consequently, Common Lisp supports both lexical and dynamic scope.

Macros

A *macro* in Lisp superficially resembles a function in usage. However, rather than representing an expression which is evaluated, it represents a transformation of the program source code. The macro gets the source it surrounds as arguments, binds them to its parameters and computes a new source form. This new form can also use a macro. The macro expansion is repeated until the new source form does not use a macro. The final computed form is the source code executed at runtime.

Typical uses of macros in Lisp:

- new control structures (example: looping constructs, branching constructs)
- scoping and binding constructs
- simplified syntax for complex and repeated source code
- top-level defining forms with compile-time side-effects
- data-driven programming
- embedded domain specific languages (examples: SQL, HTML, Prolog)

Various standard Common Lisp features also need to be implemented as macros, such as:

- the standard SETF abstraction, to allow custom compile-time expansions of assignment/access operators
 - WITH-ACCESSORS, WITH-SLOTS, WITH-OPEN-FILE and other similar WITH macros
-

- Depending on implementation, `IF` or `COND` is a macro built on the other, the special operator; `WHEN` and `UNLESS` consist of macros
- The powerful `LOOP` domain-specific language

Macros are defined by the `defmacro` macro. The special operator `macrolet` allows the definition of local (lexically scoped) macros. It is also possible to define macros for symbols using `define-symbol-macro` and `symbol-macrolet`.

Paul Graham's book On Lisp describes the use of macros in Common Lisp in detail.

Example using a macro to define a new control structure

Macros allow Lisp programmers to create new syntactic forms in the language. One typical use is to create new control structures. The example macro provides an `until` looping construct. The syntax is:

```
(until test form*)
```

The macro definition for `until`:

```
(defmacro until (test &body body)
  (let ((start-tag (gensym "START"))
        (end-tag   (gensym "END")))
    ` (tagbody ,start-tag
               (when ,test (go ,end-tag))
               (progn ,@body)
               (go ,start-tag)
               ,end-tag)))
```

`tagbody` is a primitive Common Lisp special operator which provides the ability to name tags and use the `go` form to jump to those tags. The backquote ` provides a notation that provides code templates, where the value of forms preceded with a comma are filled in. Forms preceded with comma and at-sign are *spliced* in. The `tagbody` form tests the end condition. If the condition is true, it jumps to the end tag. Otherwise the provided body code is executed and then it jumps to the start tag.

An example form using above `until` macro:

```
(until (= (random 10) 0)
      (write-line "Hello"))
```

The code can be expanded using the function `macroexpand-1`. The expansion for above example looks like this:

```
(TAGBODY
 #:START1136
 (WHEN (ZEROP (RANDOM 10))
 (GO #:END1137))
 (PROGN (WRITE-LINE "hello"))
 (GO #:START1136)
 #:END1137)
```

During macro expansion the value of the variable `test` is `(= (random 10) 0)` and the value of the variable `body` is `((write-line "Hello"))`. The body is a list of forms.

Symbols are usually automatically upcased. The expansion uses the `TAGBODY` with two labels. The symbols for these labels are computed by `GENSYM` and are not interned in any package. Two `go` forms use these tags to jump to. Since `tagbody` is a primitive operator in Common Lisp (and not a macro), it will not be expanded into something

else. The expanded form uses the *when* macro, which also will be expanded. Fully expanding a source form is called *code walking*.

In the fully expanded (*walked*) form, the *when* form is replaced by the primitive *if*:

```
(TAGBODY
  #:START1136
  (IF (ZEROP (RANDOM 10))
    (PROGN (GO #:END1137))
    NIL)
  (PROGN (WRITE-LINE "hello"))
  (GO #:START1136))
#:END1137)
```

All macros must be expanded before the source code containing them can be evaluated or compiled normally. Macros can be considered functions that accept and return abstract syntax trees (Lisp S-expressions). These functions are invoked before the evaluator or compiler to produce the final source code. Macros are written in normal Common Lisp, and may use any Common Lisp (or third-party) operator available.

Variable capture and shadowing

Common Lisp macros are capable of what is commonly called *variable capture*, where symbols in the macro-expansion body coincide with those in the calling context, allowing the programmer to create macros wherein various symbols have special meaning. The term *variable capture* is somewhat misleading, because all namespaces are vulnerable to unwanted capture, including the operator and function namespace, the tagbody label namespace, catch tag, condition handler and restart namespaces.

Variable capture can introduce software defects. This happens in one of the following two ways:

- In the first way, a macro expansion can inadvertently make a symbolic reference which the macro writer assumed will resolve in a global namespace, but the code where the macro is expanded happens to provide a local, shadowing definition it which steals that reference. Let this be referred to as type 1 capture.
- The second way, type 2 capture, is just the opposite: some of the arguments of the macro are pieces of code supplied by the macro caller, and those pieces of code are written such that they make references to surrounding bindings. However, the macro inserts these pieces of code into an expansion which defines its own bindings that accidentally captures some of these references.

The Scheme dialect of Lisp provides a macro-writing system which provides the referential transparency that eliminates both types of capture problem. This type of macro system is sometimes called "hygienic", in particular by its proponents (who regard macro systems which do not automatically solve this problem as unhygienic).

In Common Lisp, macro hygiene is ensured one of two different ways.

One approach is to use gensyms: guaranteed-unique symbols which can be used in a macro-expansion without threat of capture. The use of gensyms in a macro definition is a manual chore, but macros can be written which simplify the instantiation and use of gensyms. Gensyms solve type 2 capture easily, but they are not applicable to type 1 capture in the same way, because the macro expansion cannot rename the interfering symbols in the surrounding code which capture its references. Gensyms could be used to provide stable aliases for the global symbols which the macro expansion needs. The macro expansion would use these secret aliases rather than the well-known names, so redefinition of the well-known names would have no ill effect on the macro.

Another approach is to use packages. A macro defined in its own package can simply use internal symbols in that package in its expansion. The use of packages deals with type 1 and type 2 capture.

However, packages don't solve the type 1 capture of references to standard Common Lisp functions and operators. The reason is that the use of packages to solve capture problems revolves around the use of private symbols (symbols in one package, which are not imported into, or otherwise made visible in other packages). Whereas the Common Lisp library symbols are external, and frequently imported into or made visible in user-defined packages.

The following is an example of unwanted capture in the operator namespace, occurring in the expansion of a macro:

```
;; expansion of UNTIL makes liberal use of DO
(defmacro until (expression &body body)
  ` (do () (,expression) ,@body))

;; macrolet establishes lexical operator binding for DO
(macrolet ((do (... ) ... something else ...))
  (until (= (random 10) 0) (write-line "Hello")))
```

The UNTIL macro will expand into a form which calls DO which is intended to refer to the standard Common Lisp macro DO. However, in this context, DO may have a completely different meaning, so UNTIL may not work properly.

Common Lisp solves the problem of the shadowing of standard operators and functions by forbidding their redefinition. Because it redefines the standard operator DO, the preceding is actually a fragment of non-conforming Common Lisp, which allows implementations to diagnose and reject it.

Condition system

The *condition system* is responsible for exception handling in Common Lisp. It provides *conditions*, *handlers* and *restarts*. *Conditions* are objects describing an exceptional situation (for example an error). If a *condition* is signaled, the Common Lisp system searches for a *handler* for this condition type and calls the handler. The *handler* can now search for restarts and use one of these restarts to repair the current problem. As part of a user interface (for example of a debugger), these restarts can also be presented to the user, so that the user can select and invoke one of the available restarts. Since the condition handler is called in the context of the error (without unwinding the stack), full error recovery is possible in many cases, where other exception handling systems would have already terminated the current routine. The debugger itself can also be customized or replaced using the *DEBUGGER-HOOK* dynamic variable.

In the following example (using Symbolics Genera) the user tries to open a file in a Lisp function *test* called from the Read-Eval-Print-LOOP (REPL), when the file does not exist. The Lisp system presents four restarts. The user selects the *Retry OPEN using a different pathname* restart and enters a different pathname (lispm-init.lisp instead of lispm-int.lisp). The user code does not contain any error handling code. The whole error handling and restart code is provided by the Lisp system, which can handle and repair the error without terminating the user code.

```
Command: (test ">zippy>lispm-int.lisp")

Error: The file was not found.
      For lispm:>zippy>lispm-int.lisp.newest

LMFS:OPEN-LOCAL-LMFS-1
      Arg 0: #P"lispm:>zippy>lispm-int.lisp.newest"

s-A, <Resume>: Retry OPEN of lispm:>zippy>lispm-int.lisp.newest
s-B:           Retry OPEN using a different pathname
s-C, <Abort>:  Return to Lisp Top Level in a TELNET server
```

```
s-D:           Restart process TELNET terminal

-> Retry OPEN using a different pathname
Use what pathname instead [default lispm:>zippy>lispm-int.lisp.newest]:
lispm:>zippy>lispm-init.lisp.newest

...the program continues
```

Common Lisp Object System (CLOS)

Common Lisp includes a toolkit for object-oriented programming, the Common Lisp Object System or CLOS, which is one of the most powerful object systems available in any language. For example Peter Norvig explains how many Design Patterns are simpler to implement in a dynamic language with the features of CLOS (Multiple Inheritance, Mixins, Multimethods, Metaclasses, Method combinations, etc).^[8] Several extensions to Common Lisp for object-oriented programming have been proposed to be included into the ANSI Common Lisp standard, but eventually CLOS was adopted as the standard object-system for Common Lisp. CLOS is a dynamic object system with multiple dispatch and multiple inheritance, and differs radically from the OOP facilities found in static languages such as C++ or Java. As a dynamic object system, CLOS allows changes at runtime to generic functions and classes. Methods can be added and removed, classes can be added and redefined, objects can be updated for class changes and the class of objects can be changed.

CLOS has been integrated into ANSI Common Lisp. Generic Functions can be used like normal functions and are a first-class data type. Every CLOS class is integrated into the Common Lisp type system. Many Common Lisp types have a corresponding class. There is more potential use of CLOS for Common Lisp. The specification does not say whether conditions are implemented with CLOS. Pathnames and streams could be implemented with CLOS. These further usage possibilities of CLOS for ANSI Common Lisp are not part of the standard. Actual Common Lisp implementations are using CLOS for pathnames, streams, input/output, conditions, the implementation of CLOS itself and more.

Compiler and interpreter

Several implementations of earlier Lisp dialects provided both an interpreter and a compiler. Unfortunately often the semantics were different. These earlier Lisps implemented lexical scoping in the compiler and dynamic scoping in the interpreter. Common Lisp requires that both the interpreter and compiler use lexical scoping by default. The Common Lisp standard describes both the semantics of the interpreter and a compiler. The compiler can be called using the function *compile* for individual functions and using the function *compile-file* for files. Common Lisp allows type declarations and provides ways to influence the compiler code generation policy. For the latter various optimization qualities can be given values between 0 (not important) and 3 (most important): *speed*, *space*, *safety*, *debug* and *compilation-speed*.

There is also a function to evaluate Lisp code: *eval*. *eval* takes code as pre-parsed s-expressions and not, like in some other languages, as text strings. This way code can be constructed with the usual Lisp functions for constructing lists and symbols and then this code can be evaluated with *eval*. Several Common Lisp implementations (like Clozure CL and SBCL) are implementing *eval* using their compiler. This way code is compiled, even though it is evaluated using the function *eval*.

The file compiler is invoked using the function *compile-file*. The generated file with compiled code is called a *fasl* (from *fast load*) file. These *fasl* files and also source code files can be loaded with the function *load* into a running Common Lisp system. Depending on the implementation, the file compiler generates byte-code (for example for the Java Virtual Machine), C language code (which then is compiled with a C compiler) or, directly, native code.

Common Lisp implementations can be used interactively, even though the code gets fully compiled. The idea of an Interpreted language thus does not apply for interactive Common Lisp.

The language makes distinction between read-time, compile-time, load-time and run-time, and allows user code to also make this distinction to perform the wanted type of processing at the wanted step.

Some special operators are provided to especially suit interactive development; for instance, DEFVAR will only assign a value to its provided variable if it wasn't already bound, while DEFPARAMETER will always perform the assignment. This distinction is useful when interactively evaluating, compiling and loading code in a live image.

Some features are also provided to help writing compilers and interpreters. Symbols consist of first-level objects and are directly manipulable by user code. The PROGV special operator allows to create lexical bindings programmatically, while packages are also manipulable. The Lisp compiler itself is available at runtime to compile files or individual functions. These make it easy to use Lisp as an intermediate compiler or interpreter for another language.

Code examples

Birthday paradox

The following program calculates the smallest number of people in a room for whom the probability of completely unique birthdays is less than 50% (the so-called birthday paradox, where for 1 person the probability is obviously 100%, for 2 it is 364/365, etc.). (Answer = 23.)

```
(defconstant +year-size+ 365)

(defun birthday-paradox (probability number-of-people)
  (let ((new-probability (* (/ (- +year-size+ number-of-people)
                                +year-size+)
                            probability)))
    (if (< new-probability 0.5)
        (1+ number-of-people)
        (birthday-paradox new-probability (1+ number-of-people)))))
```

Calling the example function using the REPL (Read Eval Print Loop):

```
CL-USER > (birthday-paradox 1.0 1)
23
```

Sorting a list of person objects

We define a class PERSON and a method for displaying the name and age of a person. Next we define a group of persons as a list of PERSON objects. Then we iterate over the sorted list.

```
(defclass person ()
  ((name :initarg :name :accessor person-name)
   (age :initarg :age :accessor person-age))
  (:documentation "The class PERSON with slots NAME and AGE.))

(defmethod display ((object person) stream)
  "Displaying a PERSON object to an output stream."
  (with-slots (name age) object
    (format stream "~a (~a)" name age)))
```

```
(defparameter *group*
  (list (make-instance 'person :name "Bob" :age 33)
        (make-instance 'person :name "Chris" :age 16)
        (make-instance 'person :name "Ash" :age 23)))
"A list of PERSON objects.")

(dolist (person (sort (copy-list *group*)
                      #'>
                      :key #'person-age))
  (display person *standard-output*)
  (terpri))
```

It prints the three names with descending age.

```
Bob (33)
Ash (23)
Chris (16)
```

Exponentiating by squaring

Use of the LOOP macro is demonstrated:

```
(defun power (x n)
  (loop with result = 1
        while (plusp n)
        when (oddp n) do (setf result (* result x))
        do (setf x (* x x)
                 n (truncate n 2))
        finally (return result)))
```

Example use:

```
CL-USER > (power 2 200)
1606938044258990275541962092341162602522202993782792835301376
```

Compare with the built in exponentiation:

```
CL-USER > (= (expt 2 200) (power 2 200))
T
```

Find the list of available shells

WITH-OPEN-FILE is a macro that opens a file and provides a stream. When the form is returning, the file is automatically closed. FUNCALL calls a function object. The LOOP collects all lines that match the predicate.

```
(defun list-matching-lines (file predicate)
  "Returns a list of lines in file, for which the predicate applied to
  the line returns T."
  (with-open-file (stream file)
    (loop for line = (read-line stream nil nil)
          while line
```

```

when (funcall predicate line)
collect it)))

```

The function AVAILABLE-SHELLS calls above function LIST-MATCHING-LINES with a pathname and an anonymous function as the predicate. The predicate returns the pathname of a shell or NIL (if the string is not the filename of a shell).

```

(defun available-shells (&optional (file #p"/etc/shells"))
  (list-matching-lines
   file
   (lambda (line)
     (and (plusp (length line))
          (char= (char line 0) #\:)
          (pathname
           (string-right-trim '(\#\space #\tab) line))))))

```

An example call using Mac OS X 10.6:

```

CL-USER > (available-shells)
(#P"/bin/bash" #P"/bin/csh" #P"/bin/ksh" #P"/bin/sh" #P"/bin/tcsh"
#P"/bin/zsh")

```

Comparison with other Lisps

Common Lisp is most frequently compared with, and contrasted to, Scheme—if only because they are the two most popular Lisp dialects. Scheme predates CL, and comes not only from the same Lisp tradition but from some of the same engineers—Guy L. Steele, with whom Gerald Jay Sussman designed Scheme, chaired the standards committee for Common Lisp.

Common Lisp is a general-purpose programming language, in contrast to Lisp variants such as Emacs Lisp and AutoLISP which are embedded extension languages in particular products. Unlike many earlier Lisps, Common Lisp (like Scheme) uses lexical variable scope by default for both interpreted and compiled code.

Most of the Lisp systems whose designs contributed to Common Lisp—such as ZetaLisp and Franz Lisp—used dynamically scoped variables in their interpreters and lexically scoped variables in their compilers. Scheme introduced the sole use of lexically scoped variables to Lisp; an inspiration from ALGOL 68 which was widely recognized as a good idea. CL supports dynamically scoped variables as well, but they must be explicitly declared as "special". There are no differences in scoping between ANSI CL interpreters and compilers.

Common Lisp is sometimes termed a *Lisp-2* and Scheme a *Lisp-1*, referring to CL's use of separate namespaces for functions and variables. (In fact, CL has *many* namespaces, such as those for go tags, block names, and loop keywords). There is a long-standing controversy between CL and Scheme advocates over the tradeoffs involved in multiple namespaces. In Scheme, it is (broadly) necessary to avoid giving variables names which clash with functions; Scheme functions frequently have arguments named `lis`, `lst`, or `lyst` so as not to conflict with the system function `list`. However, in CL it is necessary to explicitly refer to the function namespace when passing a function as an argument—which is also a common occurrence, as in the `sort` example above.

CL also differs from Scheme in its handling of boolean values. Scheme uses the special values `#t` and `#f` to represent truth and falsity. CL follows the older Lisp convention of using the symbols `T` and `NIL`, with `NIL` standing also for the empty list. In CL, *any* non-`NIL` value is treated as true by conditionals, such as `if`, whereas in Scheme all non-`#f` values are treated as true. These conventions allow some operators in both languages to serve both as predicates (answering a boolean-valued question) and as returning a useful value for further computation, but in Scheme the value `()` which is equivalent to `NIL` in Common Lisp evaluates to true in a boolean expression.

Lastly, the Scheme standards documents require tail-call optimization, which the CL standard does not. Most CL implementations do offer tail-call optimization, although often only when the programmer uses an optimization directive. Nonetheless, common CL coding style does not favor the ubiquitous use of recursion that Scheme style prefers—what a Scheme programmer would express with tail recursion, a CL user would usually express with an iterative expression in `do`, `dolist`, `loop`, or (more recently) with the `iterate` package.

Implementations

See the Category Common Lisp implementations.

Common Lisp is defined by a specification (like Ada and C) rather than by one implementation (like Perl before version 6). There are many implementations, and the standard details areas in which they may validly differ.

In addition, implementations tend to come with library packages, which provide functionality not covered in the standard. Free and open source software libraries have been created to support such features in a portable way, and are most notably found in the repositories of the Common-Lisp.net^[1] and Common Lisp Open Code Collection^[9] projects.

Common Lisp implementations may use any mix of native code compilation, byte code compilation or interpretation. Common Lisp has been designed to support incremental compilers, file compilers and block compilers. Standard declarations to optimize compilation (such as function inlining or type specialization) are proposed in the language specification. Most Common Lisp implementations compile source code to native machine code. Some implementations can create (optimized) stand-alone applications. Others compile to interpreted bytecode, which is less efficient than native code, but eases binary-code portability. There are also compilers that compile Common Lisp code to C code. The misconception that Lisp is a purely interpreted language is most likely because Lisp environments provide an interactive prompt and that code is compiled one-by-one, in an incremental way. With Common Lisp incremental compilation is widely used.

Some Unix-based implementations (CLISP, SBCL) can be used as a scripting language; that is, invoked by the system transparently in the way that a Perl or Unix shell interpreter is.^[10]

List of implementations

Commercial implementations

Allegro Common Lisp

for Microsoft Windows, FreeBSD, Linux, Apple Mac OS X and various UNIX variants. Allegro CL provides an Integrated Development Environment (IDE) (for Windows and Linux) and extensive capabilities for application delivery.

Corman Common Lisp

for Microsoft Windows.

Liquid Common Lisp

formerly called Lucid Common Lisp. Only maintenance, no new releases.

LispWorks

for Microsoft Windows, FreeBSD, Linux, Apple Mac OS X and various UNIX variants. LispWorks provides an Integrated Development Environment (IDE) (available for all platforms) and extensive capabilities for application delivery.

Open Genera

for DEC Alpha.

Scieneer Common Lisp

which is designed for high-performance scientific computing.

Freely redistributable implementations

Armed Bear Common Lisp^[11]

A CL implementation that runs on the Java Virtual Machine.^[12] It includes a compiler to Java byte code, and allows access to Java libraries from CL. It was formerly just a component of the Armed Bear J Editor^[13].

CLISP

A bytecode-compiling implementation, portable and runs on a number of Unix and Unix-like systems (including Mac OS X), as well as Microsoft Windows and several other systems.

Clozure CL (CCL)

Originally a free and open source fork of Macintosh Common Lisp. As that history implies, CCL was written for the Macintosh, but Clozure CL now runs on Mac OS X, FreeBSD, Linux, Solaris and Windows. 32 and 64 bit x86 ports are supported on each platform. Additionally there are Power PC ports for Mac OS and Linux. CCL was previously known as OpenMCL, but that name is no longer used, to avoid confusion with the open source version of Macintosh Common Lisp.

CMUCL

Originally from Carnegie Mellon University, now maintained as free and open source software by a group of volunteers. CMUCL uses a fast native-code compiler. It is available on Linux and BSD for Intel x86; Linux for Alpha; Mac OS X for Intel x86 and PowerPC; and Solaris, IRIX, and HP-UX on their native platforms.

Embeddable Common Lisp (ECL)

ECL includes a bytecode interpreter and compiler. It can also compile Lisp code to machine code via a C compiler. ECL then compiles Lisp code to C, compiles the C code with a C compiler and can then load the resulting machine code. It is also possible to embed ECL in C programs, and C code into Common Lisp programs.

GNU Common Lisp (GCL)

The GNU Project's Lisp compiler. Not yet fully ANSI-compliant, GCL is however the implementation of choice for several large projects including the mathematical tools Maxima, AXIOM and (historically) ACL2. GCL runs on Linux under eleven different architectures, and also under Windows, Solaris, and FreeBSD.

Macintosh Common Lisp

Version 5.2 for Apple Macintosh computers with a PowerPC processor running Mac OS X is open source. RMCL (based on MCL 5.2) runs on Intel-based Apple Macintosh computers using the Rosetta binary translator from Apple.

ManKai Common Lisp (MKCL)

A branch of ECL. MKCL emphasises reliability, stability and overall code quality through a heavily reworked, natively multi-threaded, runtime system. On Linux, MKCL features a fully POSIX compliant runtime system.

Movitz

Implements a Lisp environment for x86 computers without relying on any underlying OS.

Poplog

Poplog implements a version of CL, with POP-11, and optionally Prolog, and Standard ML (SML), allowing mixed language programming. For all, the implementation language is POP-11, which is compiled incrementally. It also has an integrated Emacs-like editor that communicates with the compiler.

Steel Bank Common Lisp (SBCL)

A branch from CMUCL. "Broadly speaking, SBCL is distinguished from CMU CL by a greater emphasis on maintainability."^[14] SBCL runs on the platforms CMUCL does, except HP/UX; in addition, it runs on Linux for AMD64, PowerPC, SPARC, MIPS, Windows x86^[15] and has experimental support for running on Windows AMD64. SBCL does not use an interpreter by default; all expressions are compiled to native code unless the user switches the interpreter on. The SBCL compiler generates fast native code.^[16]

Ufasoft Common Lisp

port of CLISP for windows platform with core written in C++.

Other (mostly historical) implementations

Austin Kyoto Common Lisp

an evolution of Kyoto Common Lisp

Butterfly Common Lisp

an implementation written in Scheme for the BBN Butterfly multi-processor computer

CLICC

a Common Lisp to C compiler

CLOE

Common Lisp for PCs by Symbolics

Codemist Common Lisp

used for the commercial version of the computer algebra system Axiom

ExperCommon Lisp

an early implementation for the Apple Macintosh by ExperTelligence

Golden Common Lisp

an implementation for the PC by GoldHill Inc.

Ibuki Common Lisp

a commercialized version of Kyoto Common Lisp

Kyoto Common Lisp

the first Common Lisp compiler that used C as a target language. GCL, ECL and MKCL originate from this Common Lisp implementation.

L

a small version of Common Lisp for embedded systems

Lucid Common Lisp

a once popular Common Lisp implementation for UNIX systems

Procyon Common Lisp

an implementation for Windows and Mac OS, used by Franz for their Windows port of Allegro CL

Star Sapphire Common LISP

an implementation for the PC

SubL

a variant of Common Lisp used for the implementation of the Cyc knowledge-based system

Top Level Common Lisp

an early implementation for concurrent execution

WCL

a shared library implementation

Vax Common Lisp

Digital Equipment Corporation's implementation that ran on VAX systems running VMS or ULTRIX

Lisp Machines (from Symbolics, TI and Xerox) provided implementations of Common Lisp in addition to their native Lisp dialect (Lisp Machine Lisp or InterLisp). CLOS was also available. Symbolics provides a version of Common Lisp that is based on ANSI Common Lisp.

Applications

See the Category Common Lisp software.

Common Lisp is used to develop research applications (often in Artificial Intelligence), for rapid development of prototypes or for deployed applications.

Common Lisp is used in many commercial applications, including the Yahoo! Store web-commerce site, which originally involved Paul Graham and was later rewritten in C++ and Perl.^[17] Other notable examples include:

- ACT-R, a cognitive architecture used in a large number of research projects.
- Authorizer's Assistant,^{[18][19]} a large rule-based system used by American Express, analyzing credit requests.
- Cyc, a long running project with the aim to create a knowledge-based system that provides a huge amount of common sense knowledge
- The Dynamic Analysis and Replanning Tool (DART), which is said to alone have paid back during the years from 1991 to 1995 for all thirty years of DARPA investments in AI research.
- G2^[20] from Gensym, a real-time business rules engine (BRE)
- The development environment for the Jak and Daxter video game series, developed by Naughty Dog.
- ITA Software's low fare search engine, used by travel websites such as Orbitz and Kayak.com and airlines such as American Airlines, Continental Airlines and US Airways.
- Mirai, a 3d graphics suite. It was used to animate the face of Gollum in the movie Lord of the Rings: The Two Towers.
- Prototype Verification System (PVS), a mechanized environment for formal specification and verification.
- PWGL^[21] is a sophisticated visual programming environment based on Common Lisp, used in Computer assisted composition and sound synthesis.
- RacerPro^[22], a semantic web reasoning system and information repository.
- SPIKE^[23], a scheduling system for earth or space based observatories and satellites, notably the Hubble Space Telescope.

There also exist open-source applications written in Common Lisp, such as:

- ACL2, a full-featured automated theorem prover for an applicative variant of Common Lisp.
- Axiom, a sophisticated computer algebra system.
- Maxima, a sophisticated computer algebra system.
- OpenMusic is an object-oriented visual programming environment based on Common Lisp, used in Computer assisted composition.
- Stumpwm, a tiling, keyboard driven X11 Window Manager written entirely in Common Lisp.

References

- [1] <http://common-lisp.net/>
- [2] Document page ([http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+INCITS+226-1994+\(R2004\)](http://webstore.ansi.org/RecordDetail.aspx?sku=ANSI+INCITS+226-1994+(R2004))) at ANSI website
- [3] Authorship of the Common Lisp HyperSpec (<http://www.lispworks.com/documentation/HyperSpec/Front/Help.htm#Authorship>)
- [4] Reddy, Abhishek (2008-08-22). "Features of Common Lisp" (<http://random-state.net/features-of-common-lisp.html>). .
- [5] "Unicode support" (<http://www.cliki.net/Unicode Support>). *The Common Lisp Wiki*. . Retrieved 2008-08-21.
- [6] Richard P. Gabriel, Kent M. Pitman (June 1988). "Technical Issues of Separation in Function Cells and Value Cells" (<http://www.nhplace.com/kent/Papers/Technical-Issues.html>). *Lisp and Symbolic Computation* 1 (1): 81–101..
- [7] Chapter 2 (<http://letoverlambda.com/index.cl/guest/chap2.html>) of Let Over Lambda
- [8] Peter Norvig, Design Patterns in Dynamic Programming (<http://norvig.com/design-patterns/ppframe.htm>)
- [9] <http://clocc.sourceforge.net/>
- [10] CLISP as a scripting language under Unix (<http://clisp.cons.org/impnotes/quickstart.html#quickstart-unix>)
- [11] <http://common-lisp.net/project/armedbear/>
- [12] "Armed Bear Common Lisp" (<http://common-lisp.net/project/armedbear/>). .
- [13] <http://sourceforge.net/projects/armedbear-j/>
- [14] "History and Copyright" (<http://sbcl.sourceforge.net/history.html>). *Steel Bank Common Lisp*. .
- [15] "Platform Table" (<http://www.sbcl.org/platform-table.html>). *Steel Bank Common Lisp*. .
- [16] SBCL ranks above other dynamic language implementations in the 'Computer Language Benchmark Game' (<http://shootout.alioth.debian.org/u32q/benchmark.php?test=all&lang=all>)
- [17] "In January 2003, Yahoo released a new version of the editor written in C++ and Perl. It's hard to say whether the program is no longer written in Lisp, though, because to translate this program into C++ they literally had to write a Lisp interpreter: the source files of all the page-generating templates are still, as far as I know, Lisp code." Paul Graham, Beating the Averages (<http://www.paulgraham.com/avg.html>)
- [18] Authorizer's Assistant (<http://www.aaai.org/Papers/IAAI/1989/IAAI89-031.pdf>)
- [19] American Express Authorizer's Assistant (<http://www.prenhall.com/divisions/bp/app/alter/student/useful/ch9amex.html>)
- [20] http://www.gensym.com/index.php?option=com_content&view=article&id=47&Itemid=54
- [21] <http://www2.siba.fi/PWGL/>
- [22] <http://www.racer-systems.com/>
- [23] http://www.stsci.edu/resources/software_hardware/spike/

Bibliography

A chronological list of books published (or about to be published) about Common Lisp (the language) or about programming with Common Lisp (especially AI programming).

- Guy L. Steele: *Common Lisp the Language, 1st Edition*, Digital Press, 1984, ISBN 0-932376-41-X
- Rodney Allen Brooks: *Programming in Common Lisp*, John Wiley and Sons Inc, 1985, ISBN 0-471-81888-7
- Richard P. Gabriel: *Performance and Evaluation of Lisp Systems*, The MIT Press, 1985, ISBN 0-262-57193-5, PDF (<http://www.dreamsongs.com/Files/Timrep.pdf>)
- Robert Wilensky: *Common LISPcraft*, W.W. Norton & Co., 1986, ISBN 0-393-95544-3
- Eugene Charniak, Christopher K. Riesbeck, Drew V. McDermott, James R. Meehan: *Artificial Intelligence Programming, 2nd Edition*, Lawrence Erlbaum, 1987, ISBN 0-89859-609-2
- Wendy L. Milner: *Common Lisp: A Tutorial*, Prentice Hall, 1987, ISBN 0-13-152844-0
- Deborah G. Tatar: *A Programmer's Guide to Common Lisp*, Longman Higher Education, 1987, ISBN 0-13-728940-5
- Taiichi Yuasa, Masami Hagiya: *Introduction to Common Lisp*, Elsevier Ltd, 1987, ISBN 0-12-774860-1
- Christian Queinnec, Jerome Chailloux: *Lisp Evolution and Standardization*, Ios Pr Inc., 1988, ISBN 90-5199-008-1
- Taiichi Yuasa, Richard Weyhrauch, Yasuko Kitajima: *Common Lisp Drill*, Academic Press Inc, 1988, ISBN 0-12-774861-X
- Wade L. Hennessey: *Common Lisp*, McGraw-Hill Inc., 1989, ISBN 0-07-028177-7
- Tony Hasemer, John Dominque: *Common Lisp Programming for Artificial Intelligence*, Addison-Wesley Educational Publishers Inc, 1989, ISBN 0-201-17579-7

- Sonya E. Keene: *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Addison-Wesley, 1989, ISBN 0-201-17589-4
- David Jay Steele: *Golden Common Lisp: A Hands-On Approach*, Addison Wesley, 1989, ISBN 0-201-41653-0
- David S. Touretzky: *Common Lisp: A Gentle Introduction to Symbolic Computation*, Benjamin-Cummings, 1989, ISBN 0-8053-0492-4. Web/PDF (<http://www.cs.cmu.edu/~dst/LispBook/>)
- Christopher K. Riesbeck, Roger C. Schank: *Inside Case-Based Reasoning*, Lawrence Erlbaum, 1989, ISBN 0-89859-767-6
- Patrick Winston, Berthold Horn: *Lisp, 3rd Edition*, Addison-Wesley, 1989, ISBN 0-201-08319-1, Web (<http://people.csail.mit.edu/phw/Books/LISPBACK.HTML>)
- Gerard Gazdar, Chris Mellish: *Natural Language Processing in LISP: An Introduction to Computational Linguistics*, Addison-Wesley Longman Publishing Co., 1990, ISBN 0-201-17825-7
- Patrick R. Harrison: *The Common Lisp and Artificial Intelligence*, Prentice Hall PTR, 1990, ISBN 0-13-155243-0
- Timothy Koschmann: *The Common Lisp Companion*, John Wiley & Sons, 1990, ISBN 0-471-50308-8
- Molly M. Miller, Eric Benson: *Lisp Style & Design*, Digital Press, 1990, ISBN 1-55558-044-0
- Guy L. Steele: *Common Lisp the Language, 2nd Edition*, Digital Press, 1990, ISBN 1-55558-041-6, Web (<http://www.cs.cmu.edu/Groups/AI/html/cltl/cltl2.html>)
- Steven L. Tanimoto: *The Elements of Artificial Intelligence Using Common Lisp*, Computer Science Press, 1990, ISBN 0-7167-8230-8
- Peter Lee: *Topics in Advanced Language Implementation*, The MIT Press, 1991, ISBN 0-262-12151-4
- John H. Riley: *A Common Lisp Workbook*, Prentice Hall, 1991, ISBN 0-13-155797-1
- Peter Norvig: *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*, Morgan Kaufmann, 1991, ISBN 1-55860-191-0, Web (<http://norvig.com/paip.html>)
- Gregor Kiczales, Jim des Rivieres, Daniel G. Bobrow: *The Art of the Metaobject Protocol*, The MIT Press, 1991, ISBN 0-262-61074-4
- Jo A. Lawless, Molly M. Miller: *Understanding CLOS: The Common Lisp Object System*, Digital Press, 1991, ISBN 0-13-717232-X
- Mark Watson: *Common Lisp Modules: Artificial Intelligence in the Era of Neural Networks and Chaos Theory*, Springer Verlag New York Inc., 1991, ISBN 0-387-97614-0
- James L. Noyes: *Artificial Intelligence with Common Lisp: Fundamentals of Symbolic and Numeric Processing*, Jones & Bartlett Pub, 1992, ISBN 0-669-19473-5
- Stuart C. Shapiro: *COMMON LISP: An Interactive Approach*, Computer Science Press, 1992, ISBN 0-7167-8218-9, Web/PDF (<http://www.cse.buffalo.edu/pub/WWW/faculty/shapiro/Commonlisp/>)
- Kenneth D. Forbus, Johan de Kleer: *Building Problem Solvers*, The MIT Press, 1993, ISBN 0-262-06157-0
- Andreas Paepcke: *Object-Oriented Programming: The CLOS Perspective*, The MIT Press, 1993, ISBN 0-262-16136-2
- Paul Graham: *On Lisp*, Prentice Hall, 1993, ISBN 0-13-030552-9, Web/PDF (<http://www.paulgraham.com/onlisp.html>)
- Paul Graham: *ANSI Common Lisp*, Prentice Hall, 1995, ISBN 0-13-370875-6
- Otto Mayer: *Programmieren in Common Lisp*, German, Spektrum Akademischer Verlag, 1995, ISBN 3-86025-710-2
- Stephen Slade: *Object-Oriented Common Lisp*, Prentice Hall, 1997, ISBN 0-13-605940-6
- Richard P. Gabriel: *Patterns of Software: Tales from the Software Community*, Oxford University Press, 1998, ISBN 0-19-512123-6, PDF (<http://www.dreamsongs.org/Files/PatternsOfSoftware.pdf>)
- Taichi Yuasa, Hiroshi G. Okuno: *Advanced Lisp Technology*, CRC, 2002, ISBN 0-415-29819-9
- David B. Lamkins: *Successful Lisp: How to Understand and Use Common Lisp*, bookfix.com, 2004. ISBN 3-937526-00-5, Web (<http://www.psg.com/~dlamkins/sl/contents.html>)

- Peter Seibel: *Practical Common Lisp*, Apress, 2005. ISBN 1-59059-239-5, Web (<http://www.gigamonkeys.com/book/>)
- Doug Hoyte: *Let Over Lambda*, Lulu.com, 2008, ISBN 1-4357-1275-7, Web (<http://letoverlambda.com/>)
- George F. Luger, William A. Stubblefield: *AI Algorithms, Data Structures, and Idioms in Prolog, Lisp and Java*, Addison Wesley, 2008, ISBN 0-13-607047-7, PDF (http://wps.aw.com/wps/media/objects/5771/5909832/PDF/Luger_0136070477_1.pdf)
- Conrad Barski: *Land of Lisp: Learn to program in Lisp, one game at a time!*, No Starch Press, 2010, ISBN 1-59327-200-6, Web (<http://www.lisperati.com/landoflisp/>)

External links

- The CLiki (<http://www.cliki.net/>), a Wiki for free and open source Common Lisp systems running on Unix-like systems.
- Common Lisp software repository (<http://www.common-lisp.net/>).
- The Common Lisp directory - information repository for all things Common Lisp (<http://www.cl-user.net/>).
- "History" (http://www.lispworks.com/documentation/HyperSpec/Body/01_ab.htm). *Common Lisp HyperSpec*.
- Lisping at JPL (<http://www.flownet.com/gat/jpl-lisp.html>)
- The Nature of Lisp (<http://www.defmacro.org/ramblings/lisp.html>) Essay that examines Lisp by comparison with XML.
- Common Lisp Implementations: A Survey (<http://common-lisp.net/~dlw/LispSurvey.html>) Survey of maintained Common Lisp implementations.

OCaml

OCaml

Paradigm(s)	multi-paradigm: imperative, functional, object-oriented
Appeared in	1996
Developer	INRIA
Stable release	4.00.0 (July 26, 2012)
Typing discipline	static, strong, inferred
Dialects	F#, JoCaml, MetaOCaml, OcamlP3I
Influenced by	Caml Light, Standard ML
Influenced	F#, Scala, ATS, Opa
Implementation language	Programming language
OS	Cross-platform
License	Q Public License (compiler) LGPL (library)
Website	caml.inria.fr/index.en.html [1]
 Objective Caml at Wikibooks	

OCaml ( /oʊ'kæməl/ *oh-KAM-əl*), originally known as **Objective Caml**, is the main implementation of the Caml programming language, created by Xavier Leroy, Jérôme Vouillon, Damien Doligez, Didier Rémy and others in 1996. OCaml extends the core Caml language with object-oriented constructs.

OCaml's toolset includes an interactive top level interpreter, a bytecode compiler, and an optimizing native code compiler. It has a large standard library that makes it useful for many of the same applications as Python or Perl, as well as robust modular and object-oriented programming constructs that make it applicable for large-scale software engineering. OCaml is the successor to Caml Light. The acronym CAML originally stood for *Categorical Abstract Machine Language*, although OCaml abandons this abstract machine.

OCaml is a free open source project managed and principally maintained by INRIA. In recent years, many new languages have drawn elements from OCaml, most notably F# and Scala.

Philosophy

ML-derived languages are best known for their static type systems and type-inferring compilers. OCaml unifies functional, imperative, and object-oriented programming under an ML-like type system. This means the program author is not required to be overly familiar with pure functional language paradigm in order to use OCaml.

OCaml's static type system can help eliminate problems at runtime. However, it also forces the programmer to conform to the constraints of the type system, which can require careful thought and close attention. A type-inferring compiler greatly reduces the need for manual type annotations (for example, the data type of variables and the signature of functions usually do not need to be explicitly declared, as they do in Java). Nonetheless, effective use of OCaml's type system can require some sophistication on the part of the programmer.

OCaml is perhaps most distinguished from other languages with origins in academia by its emphasis on performance. Firstly, its static type system renders runtime type mismatches impossible, and thus obviates runtime type and safety checks that burden the performance of dynamically typed languages, while still guaranteeing runtime safety (except when array bounds checking is turned off, or when certain type-unsafe features like serialization are

used; these are rare enough that avoiding them is quite possible in practice).

Aside from type-checking overhead, functional programming languages are, in general, challenging to compile to efficient machine language code, due to issues such as the funarg problem. In addition to standard loop, register, and instruction optimizations, OCaml's optimizing compiler employs static program analysis techniques to optimize value boxing and closure allocation, helping to maximize the performance of the resulting code even if it makes extensive use of functional programming constructs.

Xavier Leroy has stated that "OCaml delivers at least 50% of the performance of a decent C compiler",^[2] but a direct comparison is impossible. Some functions in the OCaml standard library are implemented with faster algorithms than equivalent functions in the standard libraries of other languages. For example, the implementation of set union in the OCaml standard library is asymptotically faster than the equivalent function in the standard libraries of imperative languages (e.g. C++, Java) because the OCaml implementation exploits the immutability of sets in order to reuse parts of input sets in the output (persistence).

Features

OCaml features: a static type system, type inference, parametric polymorphism, tail recursion, pattern matching, first class lexical closures, functors (parametric modules), exception handling, and incremental generational automatic garbage collection.

OCaml is particularly notable for extending ML-style type inference to an object system in a general-purpose language. This permits structural subtyping, where object types are compatible, if their method signatures are compatible, regardless of their declared inheritance; an unusual feature in statically typed languages.

A foreign function interface for linking to C primitives is provided, including language support for efficient numerical arrays in formats compatible with both C and FORTRAN. OCaml also supports the creation of libraries of OCaml functions that can be linked to a "main" program in C, so that one could distribute an OCaml library to C programmers who have no knowledge nor installation of OCaml.

The OCaml distribution contains:

- An extensible parser and macro language named Camlp4, which permits the syntax of OCaml to be extended or even replaced
- Lexer and parser tools called ocamllex and ocaml yacc
- Debugger that supports stepping backwards to investigate errors
- Documentation generator
- Profiler — for measuring performance
- Numerous general-purpose libraries

The native code compiler is available for many platforms, including Unix, Microsoft Windows, and Apple Mac OS X. Excellent portability is ensured through native code generation support for major architectures: IA-32, IA-64, AMD64, HP/PA; PowerPC, SPARC, Alpha, MIPS, and StrongARM.

OCaml bytecode and native code programs can be written in a multithreaded style, with preemptive context switching. However, because the garbage collector of the INRIA OCaml system (which is the only currently available full implementation of the language) is not designed for concurrency, symmetric multiprocessing is not supported.^[3] OCaml threads in the same process execute by time sharing only. There are however several libraries for distributed computing such as Functory^[4] and ocamlnet^[5]/Plasma^[6] (blog^[7]).

Code examples

Snippets of OCaml code are most easily studied by entering them into the "top-level". This is an interactive OCaml session that prints the inferred types of resulting or defined expressions. The OCaml top-level is started by simply executing the OCaml program:

```
$ ocaml  
Objective Caml version 3.09.0  
#
```

Code can then be entered at the "#" prompt. For example, to calculate $1+2*3$:

```
# 1 + 2 * 3;;  
- : int = 7
```

OCaml infers the type of the expression to be "int" (a machine-precision integer) and gives the result "7".

Hello World

The following program "hello.ml":

```
print_endline "Hello World!"
```

can be compiled into a bytecode executable:

```
$ ocamlc hello.ml -o hello
```

or compiled into an optimized native-code executable:

```
$ ocamlopt hello.ml -o hello
```

and executed:

```
$ ./hello  
Hello World!  
$
```

Summing a list of integers

Lists are one of the most fundamental datatypes in OCaml. The following code example defines a *recursive* function *sum* that accepts one argument *xs*. (Notice the keyword **rec**). The function recursively iterates over a given list and provides a sum of integer elements. The *match* statement has similarities to C's switch element, though it is much more general.

```
let rec sum xs =  
  match xs with  
    | []          -> 0  
    | x :: xs'   -> x + sum xs'  
  
# sum [1;2;3;4;5];;  
- : int = 15
```

Another way is to use standard fold function that works with lists.

```
let sum xs =  
  List.fold_left (+) 0 xs
```

```
# sum [1;2;3;4;5];;
- : int = 15
```

Quicksort

OCaml lends itself to the concise expression of recursive algorithms. The following code example implements an algorithm similar to quicksort that sorts a list in increasing order.

```
let rec qsort = function
| [] -> []
| pivot :: rest ->
  let is_less x = x < pivot in
  let left, right = List.partition is_less rest in
  qsort left @ [pivot] @ qsort right
```

Birthday paradox

The following program calculates the smallest number of people in a room for whom the probability of completely unique birthdays is less than 50% (the so-called birthday paradox, where for 1 person the probability is obviously 100%, for 2 it is 364/365, etc.) (answer = 23).

```
let year_size = 365.

let rec birthday_paradox prob people =
  let prob' = (year_size -. float people) /. year_size *. prob in
  if prob' < 0.5 then
    Printf.printf "answer = %d\n" (people+1)
  else
    birthday_paradox prob' (people+1) ;

birthday_paradox 1.0 1
```

Church numerals

The following code defines a Church encoding of natural numbers, with successor (succ) and addition (add). A Church numeral

n

is a higher-order function that accepts a function

f

and a value

x

and applies

f

to

x

exactly

n

times. To convert a Church numeral from a functional value to a string, we pass it a function that prepends the string

"S"

to its input and the constant string

"0"

```
let zero f x = x
let succ n f x = f (n f x)
let one = succ zero
let two = succ (succ zero)
let add n1 n2 f x = n1 f (n2 f x)
let to_string n = n (fun k -> "S" ^ k) "0"
let _ = to_string (add (succ two) two)
```

Arbitrary-precision factorial function (libraries)

A variety of libraries are directly accessible from OCaml. For example, OCaml has a built-in library for arbitrary precision arithmetic. As the factorial function grows very rapidly, it quickly overflows machine-precision numbers (typically 32- or 64-bits). Thus, factorial is a suitable candidate for arbitrary-precision arithmetic.

In OCaml, the Num module provides arbitrary-precision arithmetic and can be loaded into a running top-level using:

```
# #load "nums.cma";;
# open Num;;
```

The factorial function may then be written using the arbitrary-precision numeric operators `=/`, `*/` and `-/`:

```
# let rec fact n =
  if n =/ Int 0 then Int 1 else n */ fact(n -/ Int 1);
val fact : Num.num -> Num.num = <fun>
```

This function can compute much larger factorials, such as 120!:

The cumbersome syntax for Num operations can be alleviated thanks to the camlp4 syntax extension called Delimited overloading [8].

Triangle (graphics)

The following program "simple.ml" renders a rotating triangle in 2D using OpenGL:

```
let () =
  ignore( Glut.init Sys.argv );
  Glut.initDisplayMode ~double_buffer:true ();
  ignore (Glut.createWindow ~title:"OpenGL Demo");
  let angle t = 10. *. t *. t in
  let render () =
    GlClear.clear [ `color ];
    GlMat.load_identity ();
    GlMat.rotate ~angle: (angle (Sys.time ())) ~z:1. ();
    GlDraw.begins `triangles;
    List.iter GlDraw.vertex2 [-1., -1.; 0., 1.; 1., -1.];
    GlDraw.ends ();
    Glut.swapBuffers () in
  GlMat.mode `modelview;
  Glut.displayFunc ~cb:render;
  Glut.idleFunc ~cb:(Some Glut.postRedisplay);
  Glut.mainLoop ()
```

The LablGL bindings to OpenGL are required. The program may then be compiled to bytecode with:

```
$ ocamlc -I +lablGL lablglut.cma lablql.cma simple.ml -o simple
```

or to nativecode with:

```
$ ocamlopt -I +lablGL lablglut.cmxa lablgl.cmxa simple.ml -o simple
```

and run.

\$./simple

Far more sophisticated, high-performance 2D and 3D graphical programs are easily developed in OCaml. Thanks to the use of OpenGL, the resulting programs are not only succinct and efficient, but also cross-platform, compiling without any changes on all major platforms.

Fibonacci Sequence

The following code calculates the Fibonacci sequence of a number n inputed. It uses tail recursion and pattern matching.

```
let rec fib_aux n a b =
  match n with
  | 0 -> a
  | _ -> fib_aux (n - 1) (a + b) a
let fib n = fib_aux n 0 1
```

Derived languages

MetaOCaml

MetaOCaml^[9] is a multi-stage programming extension of OCaml enabling incremental compiling of new machine code during runtime. Under certain circumstances, significant speedups are possible using multi-stage programming, because more detailed information about the data to process is available at runtime than at the regular compile time, so the incremental compiler can optimize away many cases of condition checking etc.

As an example: if at compile time it is known that a certain power function

```
x -> x^n
```

is needed very frequently, but the value of

```
n
```

is known only at runtime, you can use a two-stage power function in MetaOCaml:

```
let rec power n x =
  if n = 0
  then .<1>.
  else
    if even n
    then sqr (power (n/2) x)
    else .<.~x *. ~(power (n-1) x)>.
```

As soon as you know

```
n
```

at runtime, you can create a specialized and very fast power function:

```
.<fun x -> .~(power 5 .<x>.)>.
```

The result is:

```
fun x_1 -> (x_1 *
  let y_3 =
    let y_2 = (x_1 * 1)
    in (y_2 * y_2)
  in (y_3 * y_3))
```

The new function is automatically compiled.

Other derived languages

- AtomCaml^[10] provides a synchronization primitive for atomic (transactional) execution of code.
- Emily^[11] is a subset of OCaml that uses a design rule verifier to enforce object-capability [security] principles.
- F# is a Microsoft .NET language based on OCaml.
- Fresh OCaml^[12] facilitates the manipulation of names and binders.
- GCaml^[13] adds extensional polymorphism to OCaml, thus allowing overloading and type-safe marshalling.
- JoCaml integrates constructions for developing concurrent and distributed programs.
- OCamlDuce^[14] extends OCaml with features such as XML expressions and regular-expression types.
- OCamlP3l^[15] is a parallel programming system based on OCaml and the P3L language

Software written in OCaml

- XCP – The Xen Cloud Platform, an open source toolstack for the Xen Virtual Machine Hypervisor.
- FFTW – a software library for computing discrete Fourier transforms. Several C routines have been generated by an OCaml program named

```
genfft
```

- .
- Unison – a file synchronization program to synchronize files between two directories.
- Galax^[16] – an open source XQuery implementation.
- Mldonkey – a peer to peer client based on the EDonkey network.
- GeneWeb – free open source multi-platform genealogy software.
- The haXe compiler – a free open source compiler for the haXe programming language.
- Frama-c – a framework for C programs analysis.
- Liquidsoap – Liquidsoap is the audio stream generator of the Savonet project, notably used for generating the stream of netradios. [17]
- Coccinelle – Coccinelle is a program matching and transformation engine that provides the SmPL language (Semantic Patch Language) for specifying desired matches and transformations in C code. [18]
- CSIsat – a Tool for LA+EUF Interpolation.
- Orpie – a command-line RPN calculator
- Coq – a formal proof management system.
- Ocsigen – web development framework
- Mirage^[19] – operating system for constructing secure, high-performance, reliable network applications across a variety of cloud computing and mobile platforms
- Opa – an open source programming language for web development.
- CIL^[20] – CIL is a front-end for the C programming language that facilitates program analysis and transformation.
- Pfff^[21] – Pfff is a set of tools and APIs developed by Facebook.

The private trading firm Jane Street Capital has adopted OCaml as its preferred language. [22]

References

- [1] <http://caml.inria.fr/index.en.html>
- [2] Linux Weekly News (<http://lwn.net/Articles/19378/>).
- [3] Xavier Leroy's "standard lecture" on threads (<http://mirror.ocamlcore.org/caml.inria.fr/pub/ml-archives/caml-list/2002/11/64c14acb90cb14bedb2cacb73338fb15.en.html>)
- [4] <http://www.lri.fr/~filliatr/functor/About.html>
- [5] <http://projects.camlcity.org/projects/ocamlnet.html>
- [6] <http://plasma.camlcity.org/plasma/index.html>
- [7] <http://blog.camlcity.org/blog/plasma3.html>
- [8] <http://pa-do.forge.ocamlcore.org/>
- [9] MetaOCaml (<http://www.metaocaml.org/>)
- [10] <http://www.cs.washington.edu/homes/miker/atomcaml/>
- [11] <http://wiki.erights.org/wiki/Emily>
- [12] <http://www.fresh-ocaml.org/>
- [13] <http://www.ylis.s.u-tokyo.ac.jp/~furuse/gcaml/>
- [14] <http://www.cduce.org/ocaml>
- [15] <http://www.pps.jussieu.fr/~dicosmo/ocampl3l/index.html>
- [16] <http://galax.sourceforge.net/about.html>
- [17] <http://savonet.sourceforge.net/>
- [18] <http://coccinelle.lip6.fr/>
- [19] <http://www.openmirage.org/>
- [20] <http://www.cs.berkeley.edu/~necula/cil/>
- [21] <https://github.com/facebook/pfff>
- [22] <http://cacm.acm.org/magazines/2011/11/138203-ocaml-for-the-masses/fulltext>

External links

- Caml language family official website (<http://caml.inria.fr/>)
 - OCaml libraries (http://caml.inria.fr/humps/caml_latest.html)
- Try OCaml in your browser (<http://try.ocamlpro.com/>)
- OCaml tutorial for C, C++, Java and Perl programmers (<http://mirror.ocamlcore.org/ocaml-tutorial.org/>)
- Jason Hickey book (<http://files.metaprl.org/doc/ocaml-book.pdf>)
- A basic OCaml tutorial (<http://www.ocf.berkeley.edu/~mbh/tutorial/index.html>)
- A Tutorial with a practical approach. (<http://www.soton.ac.uk/~fangohr/software/ocamltutorial/index.html>)
- OCamlcore Planet (<http://planet.ocamlcore.org/>) aggregation of people and institutional feeds about OCaml.
- OCamlForge (<http://forge.ocamlcore.org/>) a free service to Open Source OCaml developers offering easy access to the best in source control management, mailing lists, bug tracking, message boards/forums, task management, site hosting, permanent file archival, full backups, and total web-based administration.
- OCaml Batteries Included (<http://batteries.forge.ocamlcore.org>), a community-built standard library for OCaml
- OCaml-Java (<http://ocamljava.x9c.fr/>), OCaml for Java
- OCamlIL (<http://www.pps.jussieu.fr/~montela/ocamil/>), an OCaml compiler for Microsoft .NET
- LablGTK (<http://wwwfun.kurims.kyoto-u.ac.jp/soft/olabl/labgtk.html>) GTK+ bindings (LablGTK)
- Newest OCaml Projects on Sourceforge (http://sourceforge.net/softwaremap/trove_list.php?stquery=&sort=registration_date&sortdir=desc&offset=0&form_cat=454)
- Comparison of the speed of various languages (<http://shootout.alioth.debian.org/>) including OCaml
- MetaOCaml (<http://www.metaocaml.org/>) home page
- the GODI (<http://godi.camlcity.org/godi/index.html>) package manager for OCaml
- Ocamlwizard (<http://ocamlwizard.lri.fr>)

Prototype

Prototype-based programming

Prototype-based programming is a style of object-oriented programming in which classes are not present, and behavior reuse (known as inheritance in class-based languages) is performed via a process of cloning existing objects that serve as prototypes. This model can also be known as *classless*, *prototype-oriented* or *instance-based* programming. Delegation is the language feature that supports prototype-based programming.

The first prototype-oriented programming language was Self developed by David Ungar and Randall Smith ^[1] in the mid 1980s to research topics in object-oriented language design. Since the late 1990s, the classless paradigm has grown increasingly popular. Some current prototype-oriented languages are ECMAScript (and its implementations JavaScript, JScript and Flash's ActionScript), Cecil, NewtonScript, Io, MOO, REBOL, and Lisaac.

Comparison with class-based models

With class-based languages, the structure of objects is specified in programmer-defined types called classes. While classes define the type of data and functionality that objects will have, instances are "usable" objects based on the patterns of a particular class. In this model, classes act as collections of behavior (methods) and structure that are the same for all instances, whereas instances carry the objects' data. The role distinction is thus primarily based on a distinction between structure and behavior on the one hand, and state on the other.

Advocates of prototype-based programming often argue that class-based languages encourage a model of development that focuses first on the taxonomy and relationships between classes. In contrast, prototype-based programming is seen as encouraging the programmer to focus on the behavior of some set of examples and only later worry about classifying these objects into archetypal objects that are later used in a fashion similar to classes.^[2] As such, many prototype-based systems encourage the alteration of prototypes during run-time, whereas only very few class-based object-oriented systems (such as the dynamic object-oriented system, Common Lisp, Smalltalk, Objective-C, Python, Perl, or Ruby) allow classes to be altered during the execution of a program.

Almost all prototype-based systems are based on interpreted and dynamically typed languages. Systems based on statically typed languages are technically feasible, however. The Omega language discussed in *Prototype-Based Programming*^[3] is an example of such a system, though according to Omega's website even Omega is not exclusively static, but rather its "compiler may choose to use static binding where this is possible and may improve the efficiency of a program."

See section "Criticism" for further comparison.

Object construction

In class-based languages, a new instance is constructed through the class' constructor function, a special function that reserves a block of memory for the object's members (properties and methods) and returns a reference to that block. An optional set of constructor arguments can be passed to the function and are usually held in properties. The resulting instance will inherit all the methods and properties that were defined in the class, which acts as a kind of template from which similar typed objects can be constructed.

In prototype-based languages there are no explicit classes and objects inherit directly from other objects with whom they are linked through a property, often called `prototype` as in the case of Javascript. There are two methods of constructing new objects: *ex nihilo* ("from nothing") object creation or through *cloning* an existing object. The

former is supported through some form of object literal declarations where objects can be defined at runtime through special syntax such as `{ ... }` and passed directly to a variable. While most systems support a variety of cloning, *ex nihilo* object creation is not as prominent.^[4]

Systems that support *ex nihilo* object creation allow new objects to be created from scratch without cloning from an existing prototype. Such systems provide a special syntax for specifying the properties and behaviors of new objects without referencing existing objects. In many prototype languages there exists a root object, often called *Object*, which is set as the default prototype for all other objects created in run-time and which carries commonly needed methods such as a `toString()` function to return a description of the object as a string. One useful aspect of *ex nihilo* object creation is to ensure that a new object's slot names do not have namespace conflicts with the top-level *Object* object. (In the Mozilla JavaScript implementation, one can do this by setting a newly constructed object's `__proto__` property to null.)

Cloning refers to a process whereby a new object is constructed by copying the behavior of an existing object (its prototype). The new object then carries all the qualities of the original. From this point on, the new object can be modified. In some systems the resulting child object maintains an explicit link (via *delegation* or *resemblance*) to its prototype, and changes in the prototype cause corresponding changes to be apparent in its clone. Other systems, such as the Forth-like programming language Kevo, do not propagate change from the prototype in this fashion, and instead follow a more *concatenative* model where changes in cloned objects do not automatically propagate across descendants.^[2]

```
// Example of true prototypal inheritance style
// in JavaScript.

// "ex nihilo" object creation using the literal
// object notation {}.
var foo = {name: "foo", one: 1, two: 2};

// Another "ex nihilo" object.
var bar = {three: 3};

// Gecko and Webkit JavaScript engines can directly
// manipulate the internal prototype link.
// For the sake of simplicity, let us pretend
// that the following line works regardless of the
// engine used:
bar.__proto__ = foo; // foo is now the prototype of bar.

// If we try to access foo's properties from bar
// from now on, we'll succeed.
bar.one // Resolves to 1.

// The child object's properties are also accessible.
bar.three // Resolves to 3.

// Own properties shadow prototype properties
bar.name = "bar";
foo.name; // unaffected, resolves to "foo"
bar.name; // Resolves to "bar"
```

This example in JS 1.8.5 + (see <http://kangax.github.com/es5-compat-table/>)

```
var foo = {one: 1, two: 2};

// bar. [[ prototype ]] = foo
var bar = Object.create( foo );

bar.three = 3;

bar.one; // 1
bar.two; // 2
bar.three; // 3
```

Delegation

In prototype-based languages that use *delegation*, the language runtime is capable of dispatching the correct method or finding the right piece of data simply by following a series of delegation pointers (from object to its prototype) until a match is found. All that is required to establish this behavior-sharing between objects is the delegation pointer. Unlike the relationship between class and instance in class-based object-oriented languages, the relationship between the prototype and its offshoots does not require that the child object have a memory or structural similarity to the prototype beyond this link. As such, the child object can continue to be modified and amended over time without rearranging the structure of its associated prototype as in class-based systems. It is also important to note that not only data but also methods can be added or changed. For this reason, most prototype-based languages refer to both data and methods as "slots".

Concatenation

Under pure prototyping, which is also referred to as *concatenative* prototypes, and is exemplified in the Kevo language, there are no visible pointers or links to the original prototype from which an object is cloned. The prototype object is copied exactly, but given a different name (or reference). Behavior and attributes are simply duplicated as-is.

One advantage of this approach is that object authors can alter the copy without worrying about side-effects across other children of the parent. Another advantage is that method lookup during dispatch is much cheaper computationally than with delegation, where an exhaustive search must be made of the entire delegation chain before failure to find a method or slot can be admitted.

Disadvantages to the concatenative approach include the organizational difficulty of propagating changes through the system; if a change occurs in a prototype, it is not immediately or automatically available on its clones. However, Kevo does provide additional primitives for publishing changes across sets of objects based on their similarity (so-called *family resemblances*) rather than through taxonomic origin, as is typical in the delegation model.

Another disadvantage is that, in the most naive implementations of this model, additional memory is wasted (versus the delegation model) on each clone for the parts that have stayed the same between prototype and clone. However, it is possible to provide concatenative behavior to the programming while sharing implementation and data behind-the-scenes; such an approach is indeed followed by Kevo.^[5]

An alternative quasi-solution to the problem of clones interfering with the behavior of the parent is to provide a means whereby the potential parent is flagged as being clonable or not. In MOO, this is achieved with the "f" ("fertile") flag. Only objects with the "f" flag can be cloned. In practice, this leads to certain objects serving as *surrogate classes*; their properties are kept constant to serve as initial values for their children. These children then tend to have the "f" flag not set.

Criticism

Advocates of class-based object models who criticize prototype-based systems often have concerns that could be seen as similar to those concerns that proponents of static type systems for programming languages have of dynamic type systems (see datatype). Usually, such concerns involve: correctness, safety, predictability, efficiency and programmer unfamiliarity.

On the first three points, classes are often seen as analogous to types (in most statically typed object-oriented languages they serve that role) and are proposed to provide contractual guarantees to their instances, and to users of their instances, that they will behave in some given fashion.

Regarding efficiency, declaring classes simplifies many compiler optimizations that allow developing efficient method and instance variable lookup. For the Self language, much development time was spent on developing, compiling, and interpreting techniques to improve the performance of prototype-based systems versus class-based systems. For example, the Lisaac compiler produces code almost as fast as C. Tests have been run with an MPEG-2 codec written in Lisaac, copied from a C version. These show the Lisaac version is 1.9% slower than the C version with 37% fewer lines of code.^[6]

A common criticism made against prototype-based languages is that the community of software developers is unfamiliar with them, despite the popularity and market permeation of JavaScript. This knowledge level of prototype-based systems seems to be changing with the proliferation of JavaScript frameworks and increases in the complex use of JavaScript as "Web 2.0" matures.

Languages

- Actor-Based Concurrent Language (ABCL): ABCL/1, ABCL/R, ABCL/R2, ABCL/c+
- Agora
- Cecil
- Cel
- ColdC
- ECMAScript
 - ActionScript, used by Adobe Flash and Adobe Flex
 - E4X
 - JavaScript
 - JScript
- Falcon
- Io
- Ioke
- Lisaac
- Logtalk
- LPC
- Lua
- MOO
- Neko
- NewtonScript
- Obliq
- Object Lisp
- Omega
- OpenLaszlo
- Perl, with the Class::Prototyped module

- R, with the proto package
- REBOL
- Self
- Seph
- SmartFrog
- TADS
- Tcl with snit extension

References

- [1] <http://research.sun.com/people/randy/>
- [2] Taivalsaari, Antero. "Section 1.1". *Classes vs. Prototypes: Some Philosophical and Historical Observations*. p. 14. CiteSeerX: 10.1.1.56.4713 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.4713>).
- [3] Blaschek, Günther. "Section 2.8". *Omega: Statically Typed Prototypes*. p. 177.
- [4] Dony, Christophe; Malenfant, Jacques; Bardou, Daniel. "Section 1.2" (<http://www.lirmm.fr/~dony/postscript/proto-book.pdf>). *Classifying Prototype-based Programming Languages*. p. 17. .
- [5] Taivalsaari, Antero (1992). "Kevo, a prototype-based object-oriented programming language based on concatenation and module operations". *Technical Report Report LACIR 92-02* (University of Victoria).
- [6] "Isaac project benchmarks" (http://isaacproject.u-strasbg.fr/li_benchs.html). . Retrieved 2007-07-24.

Further reading

- James Noble (ed.), Antero Taivalsaari (ed.), Ivan Moore (ed.), ed. (1999). *Prototype-Based Programming: Concepts, Languages and Applications*. Springer-Verlag. ISBN 981-4021-25-3.
- Abadi, Martin; Luca Cardelli. *A Theory of Objects*. Springer-Verlag. ISBN 0-387-94775-2.
- Class Warfare: Classes vs. Prototypes (<http://www.laputan.org/reflection/warfare.html>), by Brian Foote
- Essential Object Oriented JavaScript (http://brianodell.net/?page_id=516), by Brian O'Dell
- Using Prototypical Objects to Implement Shared Behavior in Object Oriented Systems (<http://web.media.mit.edu/~lieber/Lieberary/OOP/Delegation/Delegation.html>), by Henry Lieberman, 1986
- Prometheus Object System (<http://www.forcix.cx/software/prometheus.html>) for Scheme

Self (programming language)

Self

Paradigm(s)	object-oriented (prototype-based)
Appeared in	1987
Designed by	David Ungar, Randall Smith
Developer	David Ungar, Randall Smith, Stanford University, Sun Microsystems
Stable release	4.4 (July 2010)
Typing discipline	dynamic, strong
Major implementations	Self
Influenced by	Smalltalk
Influenced	NewtonScript, JavaScript, Io, Cel, Agora, Squeak, Lisaac, Lua, Factor, REBOL
Website	selflanguage.org [1]

Self is an object-oriented programming language based on the concept of *prototypes*. Self was a dialect of Smalltalk, being untyped and using Just-in-time compilation as well as the prototype-based approach to objects: it was first used as an experimental test system for language design in the 1980s and 1990s. In 2006, Self was still being developed as part of the Klein project, which was a Self virtual machine written fully in Self. The latest version is 4.4, released in July 2010.

Several just-in-time compilation (JIT) techniques were pioneered and improved in Self research as they were required to allow a very high level object oriented language to perform at up to half the speed of optimized C. Much of the development of Self took place at Sun Microsystems, and the techniques they developed were later deployed for Java's HotSpot virtual machine.

At one point a version of Smalltalk was implemented in Self. Because it was able to use the JIT this also gave extremely good performance.

History

Self was designed mostly by David Ungar and Randall Smith in 1986 while working at Xerox PARC. Their objective was to push forward the state of the art in object-oriented programming language research, once Smalltalk-80 was released by the labs and began to be taken seriously by the industry. They moved to Stanford University and continued work on the language, building the first working Self compiler in 1987. At that point, focus changed to attempting to bring up an entire system for Self, as opposed to just the language.

The first public release was in 1990, and the next year the team moved to Sun Microsystems where they continued work on the language. Several new releases followed until falling largely dormant in 1995 with the 4.0 version. The 4.3 version was released in 2006 and ran on Mac OS X and Solaris. A new release, version 4.4, has been developed by a group comprising some of the original team and independent programmers and is available for Mac OS X and Linux.

Self also inspired a number of languages based on its concepts. Most notable, perhaps, was NewtonScript for the Apple Newton and JavaScript used in all modern browsers. Other examples include Io, Cel, Lisaac and Agora. The IBM Tivoli Framework's distributed object system, developed in 1990, was, at the lowest level, a prototype based object system inspired by Self.

Prototype-based programming languages

Traditional class-based OO languages are based on a deep-rooted duality:

1. Classes define the basic qualities and behaviours of objects.
2. Object instances are particular manifestations of a class.

For example, suppose objects of the `Vehicle` class have a *name* and the ability to perform various actions, such as *drive to work* and *deliver construction materials*. `Bob's car` is a particular object (instance) of the class `Vehicle`, with the name "Bob's car". In theory one can then send a message to `Bob's car`, telling it to *deliver construction materials*.

This example shows one of the problems with this approach: `Bob's car`, which happens to be a sports car, is not able to carry and deliver construction materials (in any meaningful sense), but this is a capability that `Vehicles` are modelled to have. A more useful model arises from the use of subclassing to create specializations of `Vehicle`; for example `Sports Car` and `Flatbed Truck`. Only objects of the class `Flatbed Truck` need provide a mechanism to *deliver construction materials*; sports cars, which are ill suited to that sort of work, need only *drive fast*. However, this deeper model requires more insight during design, insight that may only come to light as problems arise.

This issue is one of the motivating factors behind **prototypes**. Unless one can predict with certainty what qualities a set of objects and classes will have in the distant future, one cannot design a class hierarchy properly. All too often the program would eventually need added behaviours, and sections of the system would need to be re-designed (or refactored) to break out the objects in a different way. Experience with early OO languages like Smalltalk showed that this sort of issue came up again and again. Systems would tend to grow to a point and then become very rigid, as the basic classes deep below the programmer's code grew to be simply "wrong". Without some way to easily change the original class, serious problems could arise.

Dynamic languages such as Smalltalk allowed for this sort of change via well-known methods in the classes; by changing the class, the objects based on it would change their behaviour. However, such changes had to be done very carefully, as other objects based on the same class might be expecting this "wrong" behavior: "wrong" is often dependent on the context. (This is one form of the fragile base class problem.) Further, in languages like C++, where subclasses can be compiled separately from superclasses, a change to a superclass can actually break precompiled subclass methods. (This is another form of the fragile base class problem, and also one form of the fragile binary interface problem.)

In Self, and other prototype-based languages, the duality between classes and object instances is eliminated.

Instead of having an "instance" of an object that is based on some "class", in Self one makes a copy of an existing object, and changes it. So `Bob's car` would be created by making a copy of an existing "Vehicle" object, and then adding the *drive fast* method, modelling the fact that it happens to be a Porsche 911. Basic objects that are used primarily to make copies are known as *prototypes*. This technique is claimed to greatly simplify dynamism. If an existing object (or set of objects) proves to be an inadequate model, a programmer may simply create a modified object with the correct behavior, and use that instead. Code which uses the existing objects is not changed.

Description

Self objects are a collection of "slots". Slots are accessor methods that return values, and placing a colon after the name of a slot sets the value. For example, for a slot called "name",

```
myPerson name
```

returns the value in name, and

```
myPerson name: 'foo'
```

sets it.

Self, like Smalltalk, uses *blocks* for flow control and other duties. Methods are objects containing code in addition to slots (which they use for arguments and temporary values), and can be placed in a Self slot just like any other object: a number for example. The syntax remains the same in either case.

Note that there is no distinction in Self between fields and methods: everything is a slot. Since accessing slots via messages forms the majority of the syntax in Self, many messages are sent to "self", and the "self" can be left off (hence the name).

Basic syntax

The syntax for accessing slots is similar to that of Smalltalk. Three kinds of messages are available:

unary

```
receiver slot_name
```

binary

```
receiver + argument
```

keyword

```
receiver keyword: arg1 With: arg2
```

All messages return results, so the receiver (if present) and arguments can be themselves the result of other messages. Following a message by a period means Self will discard the returned value. For example:

```
'Hello, World!' print.
```

This is the Self version of the hello world program. The ' syntax indicates a literal string object. Other literals include numbers, blocks and general objects.

Grouping can be forced by using parentheses. In the absence of explicit grouping, the unary messages are considered to have the highest precedence followed by binary (grouping left to right) and the keywords having the lowest. The use of keywords for assignment would lead to some extra parenthesis where expressions also had keyword messages, so to avoid that Self requires that the first part of a keyword message selector start with a lowercase letter, and subsequent parts start with an uppercase letter.

```
valid: base bottom between: ligature bottom + height and: base top / scale factor.
```

can be parsed unambiguously, and means the same as:

```
valid: ((base bottom) between: ((ligature bottom) + height) and: ((base top) / (scale factor))).
```

In Smalltalk-80, the same expression would look written as:

```
valid := self base bottom between: self ligature bottom + self height and: self base top / self scale factor.
```

assuming base, ligature, height and scale were not instance variables of self but were, in fact, methods.

Making new objects

Consider a slightly more complex example:

```
labelWidget copy label: 'Hello, World!'.
```

makes a copy of the "labelWidget" object with the copy message (no shortcut this time), then sends it a message to put "Hello, World" into the slot called "label". Now to do something with it:

```
(desktop activeWindow) draw: (labelWidget copy label: 'Hello, World!').
```

In this case the (desktop activeWindow) is performed first, returning the active window from the list of windows that the desktop object knows about. Next (read inner to outer, left to right) the code we examined earlier returns the labelWidget. Finally the widget is sent into the draw slot of the active window.

Inheritance/Delegation

In theory, every Self object is a stand-alone entity. Self has neither classes nor meta-classes. Changes to a particular object don't affect any other, but in some cases it is desirable if they did. Normally an object can understand only messages corresponding to its local slots, but by having one or more slots indicating *parent* objects, an object can **delegate** any message it doesn't understand itself to the parent object. Any slot can be made a parent pointer by adding an asterisk as a suffix. In this way Self handles duties that would use inheritance in class-based languages. Delegation can also be used to implement features such as namespaces and lexical scoping.

For example, suppose an object is defined called "bank account", that is used in a simple bookkeeping application. Usually, this object would be created with the methods inside, perhaps "deposit" and "withdraw", and any data slots needed by them. This is a prototype, which is only special in the way it is used since it also happens to be a fully functional bank account.

Traits

Making a clone of this object for "Bob's account" will create a new object which starts out exactly like the prototype. In this case we have copied the slots including the methods and any data. However a more common solution is to first make a more simple object called a traits object which contains the items that one would normally associate with a class.

In this example the "bank account" object would not have the deposit and withdraw method, but would have as a parent an object that did. In this way many copies of the bank account object can be made, but we can still change the behaviour of them all by changing the slots in that root object.

How is this any different from a traditional class? Well consider the meaning of:

```
myObject parent: someOtherObject.
```

This excerpt changes the "class" of myObject at runtime by changing the value associated with the 'parent*' slot (the asterisk is part of the slot name, but not the corresponding messages). Unlike with inheritance or lexical scoping, the delegate object can be modified at runtime.

Adding slots

Objects in Self can be modified to include additional slots. This can be done using the graphical programming environment, or with the primitive '_AddSlots:'. A **primitive** has the same syntax as a normal keyword message, but its name starts with the underscore character. The _AddSlots primitive should be avoided because it is a left over from early implementations. However, we will show it in the example below because it makes the code shorter.

An earlier example was about refactoring a simple class called Vehicle in order to be able to differentiate the behaviour between cars and trucks. In Self one would accomplish this with something like this:

```
_AddSlots: (| vehicle <- (|parent* = traits clonable|) |).
```

Since the receiver of the '_AddSlots:' primitive isn't indicated, it is "self". In the case of expressions typed at the prompt, that is an object called the "lobby". The argument for '_AddSlots:' is the object whose slots will be copied over to the receiver. In this case it is a literal object with exactly one slot. The slot's name is 'vehicle' and its value is another literal object. The "<-" notation implies a second slot called 'vehicle:' which can be used to change the first slot's value.

The "=" indicates a constant slot, so there is no corresponding 'parent:'. The literal object that is the initial value of 'vehicle' includes a single slot so it can understand messages related to cloning. A truly empty object, indicated as (||) or more simply as (), cannot receive any messages at all.

```
vehicle _AddSlots: (| name <- 'automobile' |).
```

Here the receiver is the previous object, which now will include 'name' and 'name:' slots in addition to 'parent*':

```
_AddSlots: (| sportsCar <- vehicle copy |).
```

```
sportsCar _AddSlots: (| driveToWork = (some code, this is a method) |).
```

Though previously 'vehicle' and 'sportsCar' were exactly alike, now the latter includes a new slot with a method that the original doesn't have. Methods can only be included in constant slots.

```
_AddSlots: (| porsche911 <- sportsCar copy |).
```

```
porsche911 name:'Bobs Porsche'.
```

The new object 'porsche911' started out exactly like 'sportsCar', but the last message changed the value of its 'name' slot. Note that both still have exactly the same slots even though one of them has a different value.

The environment

One feature of Self is that it is based on the same sort of virtual machine system that earlier Smalltalk systems used. That is, programs are not stand-alone entities as they are in languages such as C, but need their entire memory environment in order to run. This requires that applications be shipped in chunks of saved memory known as *snapshots* or *images*. One disadvantage of this approach is that images are sometimes large and unwieldy; however, debugging an image is often simpler than debugging traditional programs because the runtime state is easier to inspect and modify. (Interestingly, the difference between source-based and image-based development is analogous to the difference between class-based and prototypical object-oriented programming.)

In addition, the environment is tailored to the rapid and continual change of the objects in the system. Refactoring a "class" design is as simple as dragging methods out of the existing ancestors into new ones. Simple tasks like test methods can be handled by making a copy, dragging the method into the copy, then changing it. Unlike traditional systems, only the changed object has the new code, and nothing has to be rebuilt in order to test it. If the method works, it can simply be dragged back into the ancestor.

Performance

Self VMs achieved performance of approximately half the speed of optimised C on some benchmarks.^[2]

This was achieved by just-in-time compilation techniques which were pioneered and improved in Self research to make a high level language perform this well.

Garbage collection

The garbage collector for Self uses generational garbage collection which segregates objects by age. By using the memory management system to record page writes a write-barrier can be maintained. This technique gives excellent performance, although after running for some time a full garbage collection can occur, taking considerable time.

Optimizations

The run time system selectively flattens call structures. This gives modest speedups in itself, but allows extensive caching of type information and multiple versions of code for different caller types. This removes the need to do many method lookups and permits conditional branch statements and hard-coded calls to be inserted- often giving C-like performance with no loss of generality at the language level, but on a fully garbage collected system.^[3]

References

- [1] <http://selflanguage.org/>
- [2] <http://research.sun.com/jtech/pubs/97-peps.ps>
- [3] <http://www.sunlabs.com/research/self/papers/chambers-thesis/thesis.ps.Z>

Further reading

- Published papers on Self (<http://selflanguage.org/documentation/published/index.html>)
- Chambers, C. (1992), *The Design and Implementation of the SELF Compiler, an Optimizing Compiler for Object-Oriented Programming Languages*, Stanford University, CiteSeerX: 10.1.1.30.1652 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.30.1652>)

External links

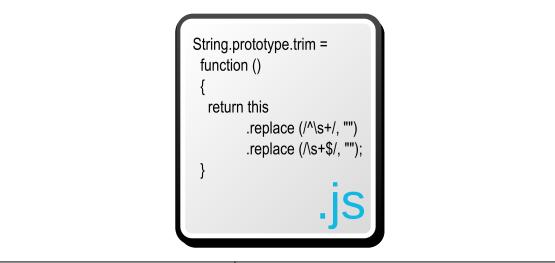
- Self Home Page (<http://selflanguage.org/>)
- Former Self Home Page at Sun Microsystems (<http://research.sun.com/self/>)
- Alternate source of papers on Self from UCSB (mirror for the Sun papers page) (<http://www.cs.ucsb.edu/~urs/oocsb/self/papers/papers.html>)
- Self resources at Cetus Links (http://www.cetus-links.org/oo_self.html)
- Merlin Project (<http://www.merlintec.com/lsi/>)
- Self ported to Linux (without many optimizations) (<http://gliebe.de/self/index.html>)
- Automated Refactoring application on sourceforge.net, written for and in Self (<http://selfguru.sourceforge.net/>)
- Gordon's Page on Self (<http://www.self-support.com/>)
- Prometheus object system on the Community Scheme Wiki (<http://community.schemewiki.org/?prometheus>)
- Video demonstrating self (<http://www.smalltalk.org.br/movies/>)
- dSelf: distributed extension to the delegation and language SELF (<http://www.ag-nbi.de/research/dself/>)

JavaScript

JavaScript

Paradigm(s)	Multi-paradigm: scripting, object-oriented (prototype-based), imperative, functional ^[1]
Appeared in	1995
Designed by	Brendan Eich
Developer	Netscape Communications Corporation, Mozilla Foundation
Stable release	1.8.5 ^[2] (March 22, 2011)
Typing discipline	dynamic, weak, duck
Major implementations	KJS, Rhino, SpiderMonkey, V8, WebKit, Carakan, Chakra
Influenced by	C, Java, Perl, Python, Scheme, Self
Influenced	ActionScript, CoffeeScript, Dart, JScript .NET, Objective-J, QML, TIScript, TypeScript
 JavaScript at Wikibooks	

JavaScript



<code>String.prototype.trim = function () { return this .replace (/^\s+/, "") .replace (/s+\$/, ""); }</code>	.js
Filename extension	.js
Internet media type	application/javascript text/javascript (obsolete) ^[3]
Uniform Type Identifier	com.netscape.javascript-source ^[4]
Type of format	Scripting language

JavaScript (sometimes abbreviated **JS**) is a programming language that is widely used to give sophisticated functionality to web pages. It is also used occasionally in non-web-related applications—for example in PDF documents, site-specific browsers, and desktop widgets. JavaScript copies many names and naming conventions from the programming language Java, but the two languages are otherwise unrelated and have very different semantics.

JavaScript is a prototype-based scripting language that is dynamic, weakly typed and has first-class functions. It uses syntax influenced by the language C. The key design principles within JavaScript are taken from the Self and Scheme programming languages.^[5] It is a multi-paradigm language, supporting object-oriented,^[6] imperative, and functional^{[1][7]} programming styles.

JavaScript was formalized in the ECMAScript language standard and is primarily used in the form of client-side JavaScript, implemented as part of a Web browser in order to create enhanced user interfaces and dynamic websites. This enables programmatic access to computational objects within a host environment. Newer and faster JavaScript VMs and frameworks built upon them (notably Node.js) have also increased the popularity of JavaScript for

server-side web applications.

History

Anyway, I know only one programming language worse than C and that is Javascript. [...] I was convinced that we needed to build-in a programming language, but the developers, Tim first, were very much opposed. It had to remain completely declarative. Maybe, but the net result is that the programming-vacuum filled itself with the most horrible kludge in the history of computing: Javascript.

Robert Cailliau^[8]

Birth at Netscape

JavaScript was originally developed in Netscape, by Brendan Eich. Battling with Microsoft over the Internet, Netscape considered their client-server solution as a distributed OS, running a portable version of Sun Microsystems' Java. Because Java was a competitor of C++ and aimed at professional programmers, Netscape also wanted a lightweight interpreted language that would complement Java by appealing to nonprofessional programmers, like Microsoft's Visual Basic.^[9] (see JavaScript and Java)

Developed under the name *Mocha*, *LiveScript* was the official name for the language when it first shipped in beta releases of Netscape Navigator 2.0 in September 1995, but it was renamed JavaScript^[10] when it was deployed in the Netscape browser version 2.0B3.^[11]

The change of name from LiveScript to JavaScript roughly coincided with Netscape adding support for Java technology in its Netscape Navigator web browser. The final choice of name caused confusion, giving the impression that the language was a spin-off of the Java programming language, and the choice has been characterized by many as a marketing ploy by Netscape to give JavaScript the cachet of what was then the hot new web programming language.^{[12][13]} It has also been claimed that the language's name is the result of a co-marketing deal between Netscape and Sun, in exchange for Netscape bundling Sun's Java runtime with its then-dominant browser.

Server-side JavaScript

Netscape introduced an implementation of the language for server-side scripting with Netscape Enterprise Server, first released in December, 1994 (soon after releasing JavaScript for browsers).^{[14][15]} Since the mid-2000s, there has been a proliferation of server-side JavaScript implementations. Node.js is one recent notable example of server-side JavaScript being used in real-world applications.^{[16][17]}

Adoption by Microsoft

JavaScript very quickly gained widespread success as a client-side scripting language for web pages. Microsoft introduced JavaScript support in its own web browser, Internet Explorer, in version 3.0, released in August 1996.^[18] Microsoft's webserver, Internet Information Server, introduced support for server-side scripting in JavaScript with release 3.0 (1996). Microsoft started to promote webpage scripting using the umbrella term Dynamic HTML.

Microsoft's JavaScript implementation was later renamed JScript to avoid trademark issues. JScript added new date methods to fix the Y2K-problematic methods in JavaScript, which were based on Java's `java.util.Date` class.^[19]

Standardization

In November 1996, Netscape announced that it had submitted JavaScript to Ecma International for consideration as an industry standard, and subsequent work resulted in the standardized version named ECMAScript.^[20]

Later developments

JavaScript has become one of the most popular programming languages on the web. Initially, however, many professional programmers denigrated the language because its target audience was web authors and other such "amateurs", among other reasons.^[21] The advent of Ajax returned JavaScript to the spotlight and brought more professional programming attention. The result was a proliferation of comprehensive frameworks and libraries, improved JavaScript programming practices, and increased usage of JavaScript outside of web browsers, as seen by the proliferation of server-side JavaScript platforms.

In January 2009, the CommonJS project was founded with the goal of specifying a common standard library mainly for JavaScript development outside the browser.^[22]

Trademark

Today, "JavaScript" is a trademark of Oracle Corporation.^[23] It is used under license for technology invented and implemented by Netscape Communications and current entities such as the Mozilla Foundation.^[24]

Features

The following features are common to all conforming ECMAScript implementations, unless explicitly specified otherwise.

Imperative and structured

JavaScript supports much of the structured programming syntax from C (e.g., `if` statements, `while` loops, `switch` statements, etc.). One partial exception is scoping: C-style block-level scoping is not supported (instead, JavaScript has function-level scoping). JavaScript 1.7, however, supports block-level scoping with the `let` keyword. Like C, JavaScript makes a distinction between expressions and statements. One syntactic difference from C is automatic semicolon insertion, in which the semicolons that terminate statements can be omitted.^[25]

Dynamic

dynamic typing

As in most scripting languages, types are associated with values, not with variables. For example, a variable `x` could be bound to a number, then later rebound to a string. JavaScript supports various ways to test the type of an object, including duck typing.^[26]

object based

JavaScript is almost entirely object-based. JavaScript objects are associative arrays, augmented with prototypes (see below). Object property names are string keys: `obj.x = 10` and `obj['x'] = 10` are equivalent, the dot notation being syntactic sugar. Properties and their values can be added, changed, or deleted at run-time. Most properties of an object (and those on its prototype inheritance chain) can be enumerated using a `for...in` loop. JavaScript has a small number of built-in objects such as `Function` and `Date`.

run-time evaluation

JavaScript includes an `eval` function that can execute statements provided as strings at run-time.

Functional

first-class functions

Functions are first-class; they are objects themselves. As such, they have properties and methods, such as `.call()` and `.bind()`,^[27] and they can be assigned to variables, passed as arguments, returned by other functions, and manipulated like any other object.^[28] Any reference to a function allows it to be invoked using the `()` operator.^[29]

nested functions and closures

"Inner" or "nested" functions are functions defined within another function. They are created each time the outer function is invoked. In addition to that, each created function forms a lexical closure: the lexical scope of the outer function, including any constants, local variables and argument values, become part of the internal state of each inner function object, even after execution of the outer function concludes.^[30]

Prototype-based

prototypes

JavaScript uses prototypes instead of classes for inheritance. It is possible to simulate many class-based features with prototypes in JavaScript.

functions as object constructors

Functions double as object constructors along with their typical role. Prefixing a function call with `new` creates a new object and calls that function with its local `this` keyword bound to that object for that invocation. The constructor's `prototype` property determines the object used for the new object's internal prototype. JavaScript's built-in constructors, such as `Array`, also have prototypes that can be modified.

functions as methods

Unlike many object-oriented languages, there is no distinction between a function definition and a method definition. Rather, the distinction occurs during function calling; a function can be called as a method. When a function is called as a method of an object, the function's local `this` keyword is bound to that object for that invocation.

Miscellaneous

run-time environment

JavaScript typically relies on a run-time environment (e.g. in a web browser) to provide objects and methods by which scripts can interact with "the outside world". In fact, it relies on the environment to provide the ability to include/import scripts (e.g. HTML `<script>` elements). (This is not a language feature per se, but it is common in most JavaScript implementations.)

variadic functions

An indefinite number of parameters can be passed to a function. The function can access them through formal parameters and also through the local `arguments` object.

array and object literals

Like many scripting languages, arrays and objects (associative arrays in other languages) can each be created with a succinct shortcut syntax. In fact, these literals form the basis of the JSON data format.

regular expressions

JavaScript also supports regular expressions in a manner similar to Perl, which provide a concise and powerful syntax for text manipulation that is more sophisticated than the built-in string functions.

Vendor-specific extensions

JavaScript is officially managed by Mozilla Foundation, and new language features are added periodically. However, only some non-Mozilla JavaScript engines support these new features:

- property getter and setter functions (also supported by WebKit, Opera,^[31] ActionScript, and Rhino)^[32]
- conditional `catch` clauses
- iterator protocol adopted from Python
- shallow generators-coroutines also adopted from Python
- array comprehensions and generator expressions also adopted from Python
- proper block scope via the `let` keyword
- array and object destructuring (limited form of pattern matching)
- concise function expressions (`function(args) expr`)
- ECMAScript for XML (E4X), an extension that adds native XML support to ECMAScript

Syntax and semantics

As of 2011, the latest version of the language is JavaScript 1.8.5. It is a superset of ECMAScript (ECMA-262) Edition 3. Extensions to the language, including partial ECMAScript for XML (E4X) (ECMA-357) support and experimental features considered for inclusion into future ECMAScript editions, are documented here.^[33]

Simple examples

There is no built-in I/O functionality in JavaScript; the runtime environment provides that. Most runtime environments have a 'console' object that can be used to print output, here is the simplest "hello world" example:

```
console.log("Hello world!");
```

A simple recursive function:

```
function factorial(n) {  
    if (n == 0) {  
        return 1;  
    }  
    return n * factorial(n - 1);  
}
```

Anonymous function (or lambda) syntax and closure example:

```
var displayClosure = function() {  
    var count = 0;  
    return function () {  
        return ++count;  
    };  
}  
  
var inc = displayClosure();  
inc(); // returns 1  
inc(); // returns 2  
inc(); // returns 3
```

Variadic function demonstration (`arguments` is a special variable).

```
var sum = function() {  
    var i, x = 0;
```

```
for (i = 0; i < arguments.length; ++i) {
    x += arguments[i];
}
return x;
}
sum(1, 2, 3); // returns 6
```

More advanced example

This sample code showcases various JavaScript features.

```
/* Finds the lowest common multiple (LCM) of two numbers */
function LCMCalculator(x, y) { // constructor function
    var checkInt = function (x) { // inner function
        if (x % 1 !== 0) {
            throw new TypeError(x + " is not an integer"); // throw an
exception
        }
        return x;
    };
    this.a = checkInt(x)
    // ^ semicolons are optional
    this.b = checkInt(y);
}

// The prototype of object instances created by a constructor is
// that constructor's "prototype" property.
LCMCcalculator.prototype = { // object literal
    constructor: LCMcalculator, // when reassigned a prototype, set
the constructor property appropriately
    gcd: function () { // method that calculates the greatest common
divisor
        // Euclidean algorithm:
        var a = Math.abs(this.a), b = Math.abs(this.b), t;
        if (a < b) {
            // swap variables
            t = b;
            b = a;
            a = t;
        }
        while (b !== 0) {
            t = b;
            b = a % b;
            a = t;
        }
        // Only need to calculate GCD once, so "redefine" this method.
        // (Actually not redefinition—it's defined on the instance
itself,
        // so that this.gcd refers to this "redefinition" instead of
LCMCcalculator.prototype.gcd.)
    }
}
```

```
// Also, 'gcd' === "gcd", this['gcd'] === this.gcd
this['gcd'] = function () {
    return a;
};

// Object property names can be specified by strings delimited by
double ("") or single ('') quotes.
"lcm" : function () {
    // Variable names don't collide with object properties, e.g.
|lcm| is not |this.lcm|.
    // not using |this.a * this.b| to avoid FP precision issues
    var lcm = this.a / this.gcd() * this.b;
    // Only need to calculate lcm once, so "redefine" this method.
    this.lcm = function () {
        return lcm;
    };
    return lcm;
},
toString: function () {
    return "LCMCalculator: a = " + this.a + ", b = " + this.b;
}
};

//define generic output function; this implementation only works for
// web browsers
function output(x) {
    document.body.appendChild(document.createTextNode(x));
    document.body.appendChild(document.createElement('br'));
}

// Note: Array's map() and forEach() are defined in JavaScript 1.6.
// They are used here to demonstrate JavaScript's inherent functional
// nature.
[[25, 55], [21, 56], [22, 58], [28, 56]].map(function (pair) { // array
    literal + mapping function
    return new LCMCalculator(pair[0], pair[1]);
}).sort(function (a, b) { // sort with this comparative function
    return a.lcm() - b.lcm();
}).forEach(function (obj) {
    output(obj + ", gcd = " + obj.gcd() + ", lcm = " + obj.lcm());
});
}
```

The following output should be displayed in the browser window.

```
LCMCalculator: a = 28, b = 56, gcd = 28, lcm = 56<br>
LCMCalculator: a = 21, b = 56, gcd = 7, lcm = 168<br>
LCMCalculator: a = 25, b = 55, gcd = 5, lcm = 275<br>
```

```
LCMCalculator: a = 22, b = 58, gcd = 2, lcm = 638<br>
```

Use in web pages

The most common use of JavaScript is to write functions that are embedded in or included from HTML pages and that interact with the Document Object Model (DOM) of the page. Some simple examples of this usage are:

- Loading new page content or submitting data to the server via AJAX without reloading the page (for example, a social network might allow the user to post status updates without leaving the page)
- Animation of page elements, fading them in and out, resizing them, moving them, etc.
- Interactive content, for example games, and playing audio and video
- Validating input values of a web form to make sure that they are acceptable before being submitted to the server.
- Transmitting information about the user's reading habits and browsing activities to various websites. Web pages frequently do this for web analytics, ad tracking, personalization or other purposes.^[34]

Because JavaScript code can run locally in a user's browser (rather than on a remote server), the browser can respond to user actions quickly, making an application more responsive. Furthermore, JavaScript code can detect user actions which HTML alone cannot, such as individual keystrokes. Applications such as Gmail take advantage of this: much of the user-interface logic is written in JavaScript, and JavaScript dispatches requests for information (such as the content of an e-mail message) to the server. The wider trend of Ajax programming similarly exploits this strength.

A JavaScript engine (also known as *JavaScript interpreter* or *JavaScript implementation*) is an interpreter that interprets JavaScript source code and executes the script accordingly. The first JavaScript engine was created by Brendan Eich at Netscape Communications Corporation, for the Netscape Navigator web browser. The engine, code-named SpiderMonkey, is implemented in C. It has since been updated (in JavaScript 1.5) to conform to ECMA-262 Edition 3. The Rhino engine, created primarily by Norris Boyd (formerly of Netscape; now at Google) is a JavaScript implementation in Java. Rhino, like SpiderMonkey, is ECMA-262 Edition 3 compliant.

A web browser is by far the most common host environment for JavaScript. Web browsers typically create "host objects" to represent the Document Object Model (DOM) in JavaScript. The web server is another common host environment. A JavaScript webserver would typically expose host objects representing HTTP request and response objects, which a JavaScript program could then interrogate and manipulate to dynamically generate web pages.

Because JavaScript is the only language that the most popular browsers share support for, it has become a target language for many frameworks in other languages, even though JavaScript was never intended to be such a language.^[35] Despite the performance limitations inherent to its dynamic nature, the increasing speed of JavaScript engines has made the language a surprisingly feasible compilation target.

Example script

Below is a minimal example of a standards-conforming web page containing JavaScript (using HTML 4.01 syntax) and the DOM:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"  
"http://www.w3.org/TR/html4/strict.dtd">  
  
<html>  
  <head><title>simple page</title></head>  
  <body>  
    <h1 id="header">This is JavaScript!</h1>  
    <script type="application/javascript">  
      document.body.appendChild(document.createTextNode('Hello World!'));  
      var h1 = document.getElementById("header"); // holds a reference to the <h1> tag  
      h1 = document.getElementsByTagName("h1")[0]; // accessing the same <h1> element
```

```
</script>
<noscript>Your browser either does not support JavaScript, or has JavaScript turned off.</noscript>
</body>
</html>
```

Compatibility considerations

Because JavaScript runs in widely varying environments, an important part of testing and debugging is to test and verify that the JavaScript works across multiple browsers.

The DOM interfaces for manipulating web pages are not part of the ECMAScript standard, or of JavaScript itself. Officially, the DOM interfaces are defined by a separate standardization effort by the W3C; in practice, browser implementations differ from the standards and from each other, and not all browsers execute JavaScript.

To deal with these differences, JavaScript authors can attempt to write standards-compliant code which will also be executed correctly by most browsers; failing that, they can write code that checks for the presence of certain browser features and behaves differently if they are not available.^[36] In some cases, two browsers may both implement a feature but with different behavior, and authors may find it practical to detect what browser is running and change their script's behavior to match.^{[37][38]} Programmers may also use libraries or toolkits which take browser differences into account.

Furthermore, scripts may not work for some users. For example, a user may:

- use an old or rare browser with incomplete or unusual DOM support,
- use a PDA or mobile phone browser which cannot execute JavaScript,
- have JavaScript execution disabled as a security precaution,
- use a speech browser due to, for example, a visual disability.

To support these users, web authors can try to create pages which degrade gracefully on user agents (browsers) which do not support the page's JavaScript. In particular, the page should remain usable albeit without the extra features that the JavaScript would have added. An alternative approach that many find preferable is to first author content using basic technologies that work in all browsers, then enhance the content for users that have JavaScript enabled. This is known as progressive enhancement.

Accessibility

Assuming that the user has not disabled its execution, client-side web JavaScript should be written to enhance the experiences of visitors with visual or physical disabilities, and certainly should avoid denying information to these visitors.^[39]

Screen readers, used by the blind and partially sighted, can be JavaScript-aware and so may access and read the page DOM after the script has altered it. The HTML should be as concise, navigable and semantically rich as possible whether the scripts have run or not. JavaScript should not be totally reliant on mouse-specific events so as to deny its benefits to users who either cannot use a mouse or who choose to favor the keyboard for whatever reason. Equally, although hyperlinks and webforms can be navigated and operated from the keyboard, accessible JavaScript should not require keyboard events either. There are device-independent events such as `onfocus` and `onchange` that are preferable in most cases.^[39]

JavaScript should not be used in a way that is confusing or disorienting to any web user. For example, using script to alter or disable the normal functionality of the browser, such as by changing the way the back-button or the refresh event work, is usually best avoided. Equally, triggering events that the user may not be aware of reduces the user's sense of control as do unexpected scripted changes to the page content.^[40]

Often the process of making a complex web page as accessible as possible becomes a nontrivial problem where issues become matters of debate and opinion, and where compromises are necessary in the end. However, user

agents and assistive technologies are constantly evolving and new guidelines and relevant information are continually being published on the web.^[39]

Security

JavaScript and the DOM provide the potential for malicious authors to deliver scripts to run on a client computer via the web. Browser authors contain this risk using two restrictions. First, scripts run in a sandbox in which they can only perform web-related actions, not general-purpose programming tasks like creating files. Second, scripts are constrained by the same origin policy: scripts from one web site do not have access to information such as usernames, passwords, or cookies sent to another site. Most JavaScript-related security bugs are breaches of either the same origin policy or the sandbox.

Cross-site vulnerabilities

A common JavaScript-related security problem is cross-site scripting, or XSS, a violation of the same-origin policy. XSS vulnerabilities occur when an attacker is able to cause a target web site, such as an online banking website, to include a malicious script in the webpage presented to a victim. The script in this example can then access the banking application with the privileges of the victim, potentially disclosing secret information or transferring money without the victim's authorization. A solution to XSS vulnerabilities is to use *HTML escaping* whenever displaying untrusted data.

Some browsers include partial protection against *reflected* XSS attacks, in which the attacker provides a URL including malicious script. However, even users of those browsers are vulnerable to other XSS attacks, such as those where the malicious code is stored in a database. Only correct design of Web applications on the server side can fully prevent XSS.

XSS vulnerabilities can also occur because of implementation mistakes by browser authors.^[41]

Another cross-site vulnerability is cross-site request forgery or CSRF. In CSRF, code on an attacker's site tricks the victim's browser into taking actions the user didn't intend at a target site (like transferring money at a bank). It works because, if the target site relies only on cookies to authenticate requests, then requests initiated by code on the attacker's site will carry the same legitimate login credentials as requests initiated by the user. In general, the solution to CSRF is to require an authentication value in a hidden form field, and not only in the cookies, to authenticate any request that might have lasting effects. Checking the HTTP Referrer header can also help.

"JavaScript hijacking" is a type of CSRF attack in which a <script> tag on an attacker's site exploits a page on the victim's site that returns private information such as JSON or JavaScript. Possible solutions include:

- requiring an authentication token in the POST and GET parameters for any response that returns private information
- using POST and never GET for requests that return private information

Misplaced trust in the client

Developers of client-server applications must recognize that untrusted clients may be under the control of attackers. Thus any secret embedded in JavaScript could be extracted by a determined adversary, and the application author cannot assume that his JavaScript runs as intended, or at all. Some implications:

- Web site authors cannot perfectly conceal how their JavaScript operates, because the code is sent to the client, and obfuscated code can be reverse-engineered.
- JavaScript form validation only provides convenience for users, not security. If a site verifies that the user agreed to its terms of service, or filters invalid characters out of fields that should only contain numbers, it must do so on the server, not only the client.
- Scripts can be selectively disabled, so JavaScript can't be relied on to prevent operations such as "save image".^[42]

- It is extremely bad practice to embed sensitive information such as passwords in JavaScript because it can be extracted by an attacker.

Browser and plugin coding errors

JavaScript provides an interface to a wide range of browser capabilities, some of which may have flaws such as buffer overflows. These flaws can allow attackers to write scripts which would run any code they wish on the user's system.

These flaws have affected major browsers including Firefox,^[43] Internet Explorer,^[44] and Safari.^[45]

Plugins, such as video players, Adobe Flash, and the wide range of ActiveX controls enabled by default in Microsoft Internet Explorer, may also have flaws exploitable via JavaScript, and such flaws have been exploited in the past.^{[46][47]}

In Windows Vista, Microsoft has attempted to contain the risks of bugs such as buffer overflows by running the Internet Explorer process with limited privileges.^[48] Google Chrome similarly limits page renderers in its own "sandbox".

Sandbox implementation errors

Web browsers are capable of running JavaScript outside of the sandbox, with the privileges necessary to, for example, create or delete files. Of course, such privileges aren't meant to be granted to code from the web.

Incorrectly granting privileges to JavaScript from the web has played a role in vulnerabilities in both Internet Explorer^[49] and Firefox.^[50] In Windows XP Service Pack 2, Microsoft demoted JScript's privileges in Internet Explorer.^[51]

Microsoft Windows allows JavaScript source files on a computer's hard drive to be launched as general-purpose, non-sandboxed programs. This makes JavaScript (like VBScript) a theoretically viable vector for a Trojan horse, although JavaScript Trojan horses are uncommon in practice.^[52] (See Windows Script Host.)

Uses outside web pages

In addition to web browsers and servers, JavaScript interpreters are embedded in a number of tools. Each of these applications provides its own object model which provides access to the host environment, with the core JavaScript language remaining mostly the same in each application.

Embedded scripting language

- Google's Chrome extensions, Opera's extensions, Apple's Safari 5 extensions, Apple's Dashboard Widgets, Microsoft's Gadgets, Yahoo! Widgets, Google Desktop Gadgets, and Serence Klipfolio are implemented using JavaScript.
- Adobe's Acrobat and Adobe Reader support JavaScript in PDF files.^[53]
- Tools in the Adobe Creative Suite, including Photoshop, Illustrator, Dreamweaver, and InDesign, allow scripting through JavaScript.
- OpenOffice.org office application suite allows for JavaScript as one of its scripting languages.
- The interactive music signal processing software Max/MSP released by Cycling '74, offers a JavaScript model of its environment for use by developers. It allows much more precise control than the default GUI-centric programming model.
- ECMAScript was included in the VRML97 standard for scripting nodes of VRML scene description files.
- Some high-end Philips universal remote panels, including TSU9600 and TSU9400, can be scripted using a JavaScript-based tool called ProntoScript.^[54]

- Sphere is an open source and cross platform computer program designed primarily to make role-playing games that use JavaScript as a scripting language.
- The open-source Re-Animator^[55] framework allows developing 2D sprite-based games using JavaScript and XML.
- Methabot is a web crawler that uses JavaScript as scripting language for custom filetype parsers and data extraction using E4X.
- The Unity game engine supports a modified JavaScript for scripting (in addition to C# and Boo) via Mono.^[56]
- DX Studio (3D engine) uses the SpiderMonkey implementation of JavaScript for game and simulation logic.^[57]
- Maxwell Render (rendering software) provides an ECMA standard based scripting engine for tasks automation.^[58]
- Google Apps Script in Google Spreadsheets and Google Sites allows users to create custom formulas, automate repetitive tasks and also interact with other Google products such as Gmail.^[59]
- Many IRC clients, like ChatZilla or XChat, use JavaScript for their scripting abilities.^{[60][61]}

Scripting engine

- Microsoft's Active Scripting technology supports JScript as a scripting language.^[62]
- The Java programming language, in version SE 6 (JDK 1.6), introduced the `javax.script` package, including a JavaScript implementation based on Mozilla Rhino. Thus, Java applications can host scripts that access the application's variables and objects, much like web browsers host scripts that access the browser's Document Object Model (DOM) for a webpage.^{[63][64]}
- The Qt C++ toolkit includes a `QtScript` module to interpret JavaScript, analogous to Java's `javax.script` package.^[65]
- JSDB^[66] (JavaScript for Databases) is an open-source JavaScript shell for Windows, Mac OS X, Linux, and Unix, which extends the Mozilla JavaScript engine with file, database, email, and network objects.
- jslibs^[67] is an open-source JavaScript shell for Windows and Linux which extends the Mozilla JavaScript engine. It has the ability to call functions in commonly used libraries like NSPR, SQLite, libTomCrypt, OpenGL, OpenAL, and librsvg.
- Late Night Software's JavaScript OSA (aka JavaScript for OSA, or JSOSA) is a freeware alternative to AppleScript for Mac OS X. It is based on the Mozilla 1.5 JavaScript implementation, with the addition of a `MacOS` object for interaction with the operating system and third-party applications.^[68]

Application platform

- ActionScript, the programming language used in Adobe Flash, is another implementation of the ECMAScript standard.
- Adobe Integrated Runtime is a JavaScript runtime that allows developers to create desktop applications.
- CA, Inc.'s AutoShell cross-application scripting environment is built on JavaScript/SpiderMonkey with preprocessor like extensions for command definitions and custom classes for various system related tasks like file i/o, operation system command invocation and redirection and COM scripting.
- GNOME Shell, the shell for the GNOME 3 desktop environment.^[69] The Seed,^[70] Gjs^[71] (from Gnome), and Kjseembed^{[72][73]} (from KDE) packages are aimed to utilize these needs.^{[74][75]}
- The Mozilla platform, which underlies Firefox, Thunderbird, and some other web browsers, uses JavaScript to implement the graphical user interface (GUI) of its various products.
- myNFC.org^[76] is a JavaScript based framework that allows developers to create applications for smart phones.
- Qt Quick's markup language (QML) is using JavaScript for the application logic, and the declarative syntax is JavaScript-like. QML has been available since Qt 4.7.
- TypeScript is a programming language based on JavaScript that adds support for optional type annotations and some other language extensions such as classes, interfaces and modules. A TS-script compiles into plain

JavaScript and can be executed in any JS host supporting ECMAScript 3 or higher. The compiler is itself written in TypeScript.

- webOS uses the WebKit implementation of JavaScript in its SDK to allow developers to create stand-alone applications solely in JavaScript.
- WinJS provides special Windows Library for JavaScript functionality in Windows 8 that enables the development of Modern style applications (formely *Metro style*) in HTML5 and JavaScript.

Development tools

Within JavaScript, access to a debugger becomes invaluable when developing large, non-trivial programs. Because there can be implementation differences between the various browsers (particularly within the Document Object Model), it is useful to have access to a debugger for each of the browsers that a web application targets.^[77]

Script debuggers are available for Internet Explorer, Firefox, Safari, Google Chrome, and Opera.^[78]

Three debuggers are available for Internet Explorer: Microsoft Visual Studio is the richest of the three, closely followed by Microsoft Script Editor (a component of Microsoft Office),^[79] and finally the free Microsoft Script Debugger which is far more basic than the other two. The free Microsoft Visual Web Developer Express^[80] provides a limited version of the JavaScript debugging functionality in Microsoft Visual Studio. Internet Explorer has included developer tools since version 8 (reached by pressing the F12 key).

Web applications within Firefox can be debugged using the Firebug add-on, or the older Venkman debugger. Firefox also has a simpler built-in Error Console, which logs and evaluates JavaScript. It also logs CSS errors and warnings.

Opera includes a set of tools called Dragonfly.^[81]

WebKit's Web Inspector includes a JavaScript debugger^[82] used in Safari, along with a modified version in Google Chrome.

Some debugging aids are themselves written in JavaScript and built to run on the Web. An example is the program JSLint, developed by Douglas Crockford who has written extensively on the language. JSLint scans JavaScript code for conformance to a set of standards and guidelines. Web development bookmarklets^[83] and Firebug Lite^[84] provide variations on the idea of the cross-browser JavaScript console.

MiniME^[85] is an open-source JavaScript minifier, obfuscator, and code-checking tool for the .NET platform.

Pretty Diff^[86] is an algorithmic diff tool written in JavaScript for comparing white space compressed JavaScript, and several other languages, to human readable forms of similar code.

SplineTech JavaScript Debugger PRO^[87] is an independent standalone JavaScript Debugging tool for Internet Explorer.

Versions

The following table is based on a history compilation forum post,^[88] jQuery author's blog post,^[89] and Microsoft's JScript version information webpage.^[90]

Version	Release date	Equivalent to	Netscape Navigator	Mozilla Firefox	Internet Explorer	Opera	Safari	Google Chrome
1.0	March 1996		2.0		3.0			
1.1	August 1996		3.0					
1.2	June 1997		4.0-4.05					
1.3	October 1998	ECMA-262 1st + 2nd edition	4.06-4.7x		4.0			
1.4			Netscape Server					
1.5	November 2000	ECMA-262 3rd edition	6.0	1.0 5.5 (JScript 5.5), 6 (JScript 5.6), 7 (JScript 5.7), 8 (JScript 5.8)	7.0	3.0-5	1.0-10.0.666	
1.6	November 2005	1.5 + array extras + array and string generics + E4X		1.5				
1.7	October 2006	1.6 + Pythonic generators [91] + iterators + let		2.0				
1.8	June 2008	1.7 + generator expressions + expression closures		3.0		11.50		
1.8.1		1.8 + native JSON support + minor updates		3.5				
1.8.2	June 22, 2009	1.8.1 + minor updates		3.6				
1.8.5	July 27, 2010	1.8.2 + ECMAScript 5 compliance		4	9	11.60		

Related languages and features

JSON, or JavaScript Object Notation, is a general-purpose data interchange format that is defined as a subset of JavaScript's literal syntax.

jQuery and Prototype are popular JavaScript libraries designed to simplify DOM-oriented client-side HTML scripting.

Mozilla browsers currently support LiveConnect, a feature that allows JavaScript and Java to intercommunicate on the web. However, Mozilla-specific support for LiveConnect is scheduled to be phased out in the future in favor of passing on the LiveConnect handling via NPAPI to the Java 1.6+ plug-in (not yet supported on the Mac as of March 2010).^[92] Most browser inspection tools, such as Firebug in Firefox, include JavaScript interpreters that can act on the visible page's DOM.

Use as an intermediate language

As JavaScript is the most widely supported programming language you can run within web browsers, it has become an intermediate language for other languages to target. This has included ports of existing languages, and the creation of new ones. Some of these include:

- Objective-J, a superset of JavaScript that compiles to standard JavaScript. It adds traditional inheritance and Smalltalk/Objective-C style dynamic dispatch and optional pseudo-static typing to JavaScript.
- Processing.js, a JavaScript port of Processing, a programming language designed to write visualizations, images, and interactive content. It allows web browsers to display animations, visual applications, games and other graphical rich content without the need for a Java applet or Flash plugin.
- CoffeeScript, an alternate syntax for JavaScript intended to be more concise and readable and adding features like array comprehensions (also available on JavaScript since version 1.7^[93]) and pattern matching. Like Objective-J,

it compiles to JavaScript. Ruby and Python have been cited as influential on CoffeeScript syntax.

- Google Web Toolkit translates a subset of Java to JavaScript.
- Quby^[94], a proprietary sand-boxed Ruby-like language by PlayMyCode used for building browser games.
- OMeta^[95], a functional language featuring pattern matching.
- Phype^[96], an open source PHP to JavaScript compiler.
- TIScript^[97], a superset of JavaScript that adds classes, namespaces, and lambda expressions.
- ClojureScript^[98], a Clojure to JavaScript compiler which is compatible with the advanced compilation mode of the Google Closure optimizing compiler.
- Parenscript^[99], a Common Lisp library that can translate into JavaScript both well-circumscribed Common Lisp code, and JavaScript rendered as "inlined" S-expressions in Common Lisp source code.
- Py2JS^[100], a subset of Python
- Pyjs^[101], a port of Google Web Toolkit to Python (translates a subset of Python to JavaScript)
- Dart is supported by Google, and compiles to its own native language in addition to a JavaScript based one
- Whalesong^[102], a Racket-to-JavaScript compiler.
- Emscripten, llvm-backend for porting native libraries to js
- Fantom a programming language that runs on jvm, .net and javascript.
- TypeScript, a free and open source programming language developed by Microsoft. It is a superset of JavaScript, and essentially adds optional static typing and class-based object oriented programming to the language.

JavaScript and Java

A common misconception is that JavaScript is similar or closely related to Java. It is true that both have a C-like syntax, the C language being their most immediate common ancestor language. They are both object-oriented, typically sandboxed (when used inside a browser), and are widely used in client-side Web applications. In addition, JavaScript was designed with Java's syntax and standard library in mind. In particular, all Java keywords were reserved in original JavaScript, JavaScript's standard library follows Java's naming conventions, and JavaScript's Math and Date objects are based on classes from Java 1.0.^[18]

JS had to "look like Java" only less so, [it had to] be Java's dumb kid brother or boy-hostage sidekick. Plus, I had to be done in ten days or something worse than JS would have happened

—Brendan Eich^[103]

However, the similarities end there. Java has static typing; JavaScript's typing is dynamic (meaning a variable can hold an object of any type and cannot be restricted). JavaScript is weakly typed ('0.0000' == 0, 0 == "", false == "", etc.) while Java is more strongly typed. Java is loaded from compiled bytecode; JavaScript is loaded as human-readable source code. Java's objects are class-based; JavaScript's are prototype-based. JavaScript also has many functional programming features based on the Scheme language.

References

- [1] Douglas Crockford (flv). *Douglas Crockford on Functional JavaScript* (http://www.blinkx.com/video/douglas-crockford-on-functional-javascript/xscZz8XhfuNQ_aaVuyUB2A) (Tech talk). blinkx. Event occurs at 2:49. . "[JavaScript] is also coincidentally the world's most popular functional programming language. JavaScript is and has always been, at least since [version] 1.2, a functional programming language."
- [2] New in JavaScript 1.8.5 | Mozilla Developer Network (https://developer.mozilla.org/en/JavaScript/New_in_JavaScript/1.8.5)
- [3] RFC 4329 (<http://www.apps.ietf.org/rfc/rfc4329.html#sec-7.1>)
- [4] "System-Declared Uniform Type Identifiers" (<http://developer.apple.com/mac/library/documentation/Miscellaneous/Reference/UTIRef/Articles/System-DeclaredUniformTypeIdentifiers.html>). *Mac OS X Reference Library*. Apple Inc.. . Retrieved 2010-03-05.
- [5] "ECMAScript Language Overview" (<http://www.ecmascript.org/es4/spec/overview.pdf>) (PDF). 2007-10-23. p. 4. . Retrieved 2009-05-03.
- [6] "ECMAScript Language Specification" (<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-262.pdf>). .
- [7] The Little JavaScripter (<http://www.crockford.com/javascript/little.html>) shows the relationship with Scheme in more detail.

- [8] wikinews:Wikinews:Story preparation/Interview with Robert Cailliau
- [9] Severance, Charles (February 2012). "Java Script: Designing a Language in 10 Days" (<http://www.computer.org/portal/web/csdl/abs/html/mags/co/2012/02/mco2012020007.htm>). *Computer* (IEEE Computer Society) **45** (2): 7–8. doi:10.1109/MC.2012.57. . Retrieved 23 April 2012.
- [10] Press release announcing JavaScript (<http://web.archive.org/web/20070916144913/http://wp.netscape.com/newsref/pr/newsrelease67.html>), "Netscape and Sun announce Javascript", PR Newswire, December 4, 1995
- [11] "TechVision: Innovators of the Net: Brendan Eich and JavaScript" (http://web.archive.org/web/20080208124612/http://wp.netscape.com/comprod/columns/techvision/innovators_be.html). Web.archive.org. Archived from the original on 2008-02-08. .
- [12] "Programming languages used on the Internet and the World Wide Web (WWW)" (http://www.webdevelopersnotes.com/basics/languages_on_the_internet.php3). Webdevelopersnotes.com. . Retrieved 2009-05-19.
- [13] "O'Reilly - Safari Books Online - 0596101996 - JavaScript: The Definitive Guide, 5th Edition" (<http://safari.oreilly.com/0596101996/javascript5-CHP-1>). Safari.oreilly.com. . Retrieved 2009-05-19.
- [14] "Server-Side JavaScript Guide" (<http://docs.oracle.com/cd/E19957-01/816-6411-10/getstart.htm>). Netscape Communications Corporation. 1998. . Retrieved 2012-04-25.
- [15] Mike Morgan (1996). "Using Netscape™ LiveWire™, Special Edition" (<http://vampire.rulez.org/onlinedoc/book/NetscapeLiveWire/ch6.htm>). Que. .
- [16] "Server-Side Javascript: Back With a Vengeance" (http://www.readwriteweb.com/archives/server-side_javascript_back_with_a_vengeance.php). *Read Write Web*. December 17, 2009. . Retrieved May 28, 2012.
- [17] "Node's goal is to provide an easy way to build scalable network programs" (<http://nodejs.org/about/>). *About Node.js*. Joyent. .
- [18] Brendan Eich (3 April 2008). "Popularity" (<http://brendaneich.com/2008/04/popularity/>). . Retrieved 2012-01-19.
- [19] <http://java.sun.com/j2se/1.4.2/docs/api/java/util/Date.html>
- [20] ECMAScript 3rd Edition specification (<http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>)
- [21] "JavaScript: The World's Most Misunderstood Programming Language" (<http://www.crockford.com/javascript/javascript.html>). Crockford.com. . Retrieved 2009-05-19.
- [22] Kris Kowal (1 December 2009). "CommonJS effort sets JavaScript on path for world domination" (<http://arstechnica.com/web/news/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination.ars>). *Ars Technica*. Condé Nast Publications. . Retrieved 18 April 2010.
- [23] "USPTO Copyright entry #75026640" (<http://tarr.uspto.gov/servlet/tarr?regser=serial&entry=75026640>). USPTO. .
- [24] "Sun Trademarks" (<http://www.sun.com/suntrademarks/>). Sun Microsystems. . Retrieved 2007-11-08.
- [25] Flanagan 2006, p. 16
- [26] Flanagan 2006, pp. 176–178
- [27] Properties of the Function Object (<http://es5.github.com/#x15.3.4-toc>), ECMAScript 5.1 (annotated version)
- [28] Flanagan 2006, p. 134
- [29] Flanagan 2006, p. 81
- [30] Flanagan 2006, p. 141
- [31] Robert Nyman, Getters And Setters With JavaScript – Code Samples And Demos (<http://robertnyman.com/2009/05/28/getters-and-setters-with-javascript-code-samples-and-demos/>), published 29 May 2009, accessed 2 January 2010.
- [32] John Resig, JavaScript Getters and Setters (<http://ejohn.org/blog/javascript-getters-and-setters/>), 18 July 2007, accessed 2 January 2010
- [33] "MDN - About this Reference" (https://developer.mozilla.org/en/JavaScript/Reference/About#JavaScript_history). Developer.mozilla.org. 2008-08-31. . Retrieved 2009-05-19.
- [34] "JavaScript tracking - Piwik" (<http://piwik.org/docs/javascript-tracking/>). Piwik. . Retrieved 31 March 2012.
- [35] Hamilton, Naomi (2008-06-31). "The A-Z of Programming Languages: JavaScript" (http://www.computerworld.com.au/article/255293/-z_programming_languages_javascript). computerworld.com.au. .
- [36] Peter-Paul Koch, Object detection (<http://www.quirksmode.org/js/support.html>)
- [37] Peter-Paul Koch, Mission Impossible - mouse position (<http://www.evolt.org/node/23335>)
- [38] Peter-Paul Koch, Browser detect (<http://www.quirksmode.org/js/detect.html>)
- [39] Flanagan 2006, pp. 262–263
- [40] "Creating Accessible JavaScript" (<http://www.webaim.org/techniques/javascript/>). WebAIM. . Retrieved 8 June 2010.
- [41] MozillaZine, Mozilla Cross-Site Scripting Vulnerability Reported and Fixed (<http://www.mozilla.org/talkback.html?article=4392>)
- [42] *Right-click "protection"? Forget about it* (<http://blog.anta.net/2008/06/17/right-click-%E2%80%93-protection%E2%80%93-forget-about-it/>). 2008-06-17. ISSN 1797-1993. . Retrieved 2008-06-17.
- [43] Mozilla Corporation, Buffer overflow in crypto.signText() (<http://www.mozilla.org/security/announce/2006/mfsa2006-38.html>)
- [44] Paul Festa, CNet, Buffer-overflow bug in IE (<http://news.com.com/2100-1001-214620.html>)
- [45] SecurityTracker.com, Apple Safari JavaScript Buffer Overflow Lets Remote Users Execute Arbitrary Code and HTTP Redirect Bug Lets Remote Users Access Files (<http://securitytracker.com/alerts/2006/Mar/1015713.html>)
- [46] SecurityFocus, Microsoft WebViewFolderIcon ActiveX Control Buffer Overflow Vulnerability (<http://www.securityfocus.com/bid/19030/info>)
- [47] Fusion Authority, Macromedia Flash ActiveX Buffer Overflow (<http://www.fusionauthority.com/security/3234-macromedia-flash-activex-buffer-overflow.htm>)

- [48] Mike Friedman, Protected Mode in Vista IE7 (<http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx>)
- [49] US CERT, Vulnerability Note VU#713878: Microsoft Internet Explorer does not properly validate source of redirected frame (<https://www.kb.cert.org/vuls/id/713878>)
- [50] Mozilla Foundation, Mozilla Foundation Security Advisory 2005-41: Privilege escalation via DOM property overrides (<http://www.mozilla.org/security/announce/2005/mfsa2005-41.html>)
- [51] Microsoft Corporation, Changes to Functionality in Microsoft Windows XP Service Pack 2: Part 5: Enhanced Browsing Security (<http://technet.microsoft.com/en-us/library/bb457150.aspx#EHA>)
- [52] For one example of a rare JavaScript Trojan Horse, see Symantec Corporation, JS.Seeker.K (http://www.symantec.com/security_response/writeup.jsp?docid=2003-100111-0931-99)
- [53] "JavaScript for Acrobat" (<http://www.adobe.com/devnet/acrobat/javascript.html>). . Retrieved 2009-08-18.
- [54] "Koninklijke Philips Electronics NV" (<http://www.pronto.philips.com/prontoscript/index.cfm?id=1422>). .
- [55] <http://www.green-eyed-monster.com/reanimator/>
- [56] "Best Of All Worlds" (<http://unity3d.com/unity/features/scripting>). unity3d.com. . Retrieved 2009-09-12.
- [57] "Technical Specification" (http://www.dxstudio.com/features_tech.aspx). dxstudio.com. . Retrieved 2009-10-20.
- [58] THINK! The Maxwell Render Resourcer Center, Scripting References (http://think.maxwellrender.com/scripting_references-269.html)
- [59] Google Apps Script, Welcome to Google Apps Script (<http://www.google.com/google-d-s/scripts/scripts.html>)
- [60] "ChatZilla! Frequently Asked Questions - 4.5. How do I write scripts?" (<http://chatzilla.hacksrus.com/faq/#scripts>). Hacksrus.com. . Retrieved 11 February 2011.
- [61] "" (<http://unborn.ludost.net/xcdscript/>). . Retrieved 11 February 2011.
- [62] Version Information (JavaScript) ([http://msdn.microsoft.com/en-us/library/s4esdbwz\(v=VS.94\).aspx](http://msdn.microsoft.com/en-us/library/s4esdbwz(v=VS.94).aspx))
- [63] "javax.script release notes" (<http://java.sun.com/javase/6/webnotes/index.html#scripting>). Java.sun.com. . Retrieved 2009-05-19.
- [64] Flanagan 2006, pp. 214 et seq
- [65] Nokia Corporation, QtScript Module (<http://doc.qt.nokia.com/4.6/qtscript.html>)
- [66] <http://www.jsdb.org/>
- [67] <http://code.google.com/p/jslibs/>
- [68] Open Scripting Architecture
- [69] "Behind the Scenes with Owen Taylor" (<http://gnomejournal.org/article/74/behind-the-scenes-with-owen-taylor>). The GNOME Journal. . Retrieved 2010-01-23.
- [70] Devel.akbkhome.com (<http://devel.akbkhome.com/seed/index.shtml>)
- [71] <http://live.gnome.org/Gjs>
- [72] <http://xmelegance.org/kjsembed/>
- [73] Xmelegance.org (<http://xmelegance.org/kjsembed/jsref/index.html>)
- [74] Gjs Links (<http://live.gnome.org/Gjs#Links>)—"code in gnome-shell" item
- [75] git.gnome.org/browse/gnome-shell/tree/js (<http://git.gnome.org/browse/gnome-shell/tree/js>) - location of gnome-shell JavaScript code in git.gnome.org repository
- [76] <http://www.myxfc.org/>
- [77] "Advanced Debugging With JavaScript" (<http://www.alistapart.com/articles/advanced-debugging-with-javascript/>). alistapart.com. 2009-02-03. . Retrieved 2010-05-28.
- [78] "The JavaScript Debugging Console" (<http://javascript.about.com/od/problemsolving/ig/JavaScript-Debugging/>). javascript.about.com. 2010-05-28. . Retrieved 2010-05-28.
- [79] JScript development in Microsoft Office 11 ([http://msdn2.microsoft.com/en-us/library/aa202668\(office.11\).aspx](http://msdn2.microsoft.com/en-us/library/aa202668(office.11).aspx)) (MS InfoPath 2003)
- [80] <http://www.microsoft.com/express/vwd/>
- [81] "Opera DragonFly" (<http://www.opera.com/dragonfly/>). Opera Software. .
- [82] "Introducing Drosera - Surfin' Safari" (<http://webkit.org/blog/61/introducing-drosera/>). Webkit.org. 2006-06-28. . Retrieved 2009-05-19.
- [83] <https://www.squarefree.com/bookmarklets/webdevel.html>
- [84] <http://getfirebug.com/lite.html>
- [85] <http://www.toptensoftware.com/minime>
- [86] <http://prettydiff.com/>
- [87] <http://www.javascript-debugger.com/>
- [88] "JavaScript - JScript - ECMAScript version history" (<http://www.webmasterworld.com/forum91/68.htm>). Webmasterworld.com. . Retrieved 2009-12-17.
- [89] John Resig. "Versions of JavaScript" (<http://ejohn.org/blog/versions-of-javascript>). Ejohn.org. . Retrieved 2009-05-19.
- [90] "Version Information (JScript)" (<http://msdn.microsoft.com/library/default.asp?url=/library/en-us/script56/html/js56jsorversioninformation.asp>). Msdn.microsoft.com. . Retrieved 2009-12-17.
- [91] https://developer.mozilla.org/en-US/docs/JavaScript/New_in_JavaScript/1.7?redirectlocale=en-US&redirectslug>New_in_JavaScript_1.7#Generators
- [92] Java.sun.com (<http://java.sun.com/javase/6/webnotes/6u10/plugin2/liveconnect/>)
- [93] New in JavaScript 1.7 ([https://developer.mozilla.org/en/New_in_JavaScript_1.7#Array_comprehensions_\(Merge_into_Array_comprehensions\)](https://developer.mozilla.org/en/New_in_JavaScript_1.7#Array_comprehensions_(Merge_into_Array_comprehensions)))

- [94] <http://www.playmycode.com/build/sandbox>
- [95] <http://tinlizzie.org/ometa/>
- [96] <https://code.google.com/p/phype/>
- [97] <https://code.google.com/p/tiscript/>
- [98] <https://github.com/clojure/clojurescript>
- [99] <http://common-lisp.net/project/parenscript>
- [100] <https://github.com/qsnake/py2js>
- [101] <http://pyjs.org/>
- [102] <http://hashcollision.org/whalesong/>
- [103] Zawinski, Jamie. "Every day I learn something new... and stupid (comment by Brendan Eich)" (<http://www.jwz.org/blog/2010/10/every-day-i-learn-something-new-and-stupid/#comment-1021>). . Retrieved 4 May 2012.

Further reading

- Bhangal, Sham; Jankowski, Tomasz (2003). *Foundation Web Design: Essential HTML, JavaScript, CSS, PhotoShop, Fireworks, and Flash*. APress L. P.. ISBN 1-59059-152-6.
- Burns, Joe; Gowney, Andree S. (2001). *JavaScript Goodies*. Pearson Education. ISBN 0-7897-2612-2.
- Duffy, Scott (2003). *How to do Everything with JavaScript*. Osborne. ISBN 0-07-222887-3.
- Flanagan, David; Ferguson, Paula (2002). *JavaScript: The Definitive Guide* (4th ed.). O'Reilly & Associates. ISBN 0-596-00048-0.
- Flanagan, David (2006). *JavaScript: The Definitive Guide* (5th ed.). O'Reilly & Associates. ISBN 0-596-10199-6.
- Goodman, Danny; Markel, Scott (2003). *JavaScript and DHTML Cookbook*. O'Reilly & Associates. ISBN 0-596-00467-2.
- Goodman, Danny; Eich, Brendan (2001). *JavaScript Bible*. John Wiley & Sons. ISBN 0-7645-3342-8.
- Harris, Andy (2001). *JavaScript Programming for the Absolute Beginner*. Premier Press. ISBN 0-7615-3410-5.
- Haverbeke, Marijn (2011). *Eloquent JavaScript*. No Starch Press. ISBN 978-1593272821.
- Heinle, Nick; Koman, Richard (1997). *Designing with JavaScript*. O'Reilly & Associates. ISBN 1-56592-300-6.
- McDuffie, Tina Spain (2003). *JavaScript Concepts & Techniques: Programming Interactive Web Sites*. Franklin, Beedle & Associates. ISBN 1-887902-69-4.
- McFarlane, Nigel (2003). *Rapid Application Development with Mozilla*. Prentice Hall Professional Technical References. ISBN 0-13-142343-6.
- Powell, Thomas A.; Schneider, Fritz (2001). *JavaScript: The Complete Reference*. McGraw-Hill Companies. ISBN 0-07-219127-9.
- Shelly, Gary B.; Cashman, Thomas J.; Dorin, William J.; Quasney, Jeffrey J. (2000). *JavaScript: Complete Concepts and Techniques*. Cambridge: Course Technology. ISBN 0-7895-6233-2.
- Watt, Andrew H.; Watt, Jonathan A.; Simon, Jinjer L. (2002). *Teach Yourself JavaScript in 21 Days*. Pearson Education. ISBN 0-672-32297-8.
- Vander Veer, Emily A. (2004). *JavaScript For Dummies* (4th ed.). Wiley Pub.. ISBN 0-7645-7659-3.

External links

- Google's Video tutorials on JS (<http://code.google.com/edu/submissions/html-css-javascript/>)
- Douglas Crockford's video lectures on JavaScript (<http://yuiblog.com/crockford/>)
- FAQ for Usenet's comp.lang.javascript (<http://jibbering.com/faq/>)
- Mozilla Developer Center
 - Mozilla's Official Documentation on JavaScript (<https://developer.mozilla.org/en/JavaScript>)
 - References for Core JavaScript versions: 1.5+ (<https://developer.mozilla.org/en/JavaScript/Reference>)
 - overview over new features in JavaScript (https://developer.mozilla.org/en/JavaScript/New_in_JavaScript)
 - List of JavaScript releases: versions 1.5+ (<https://developer.mozilla.org/en/JavaScript/Reference/About>)
 - Re-Introduction to JavaScript (https://developer.mozilla.org/en/A_re-introduction_to_JavaScript)

- Eloquent JavaScript (<http://eloquentjavascript.net/>) by Marijn Haverbeke—a free, Creative Commons-licensed eBook
- JavaScript (<http://dev.opera.com/articles/javascript/>)—Opera Developer Community
- JavaScript Libraries Comparison Matrix Tool (<http://www.html5libs.com/>)
- List of languages that compile to JS (List-of-languages-that-compile-to-JS/)

The code that writes code

Metaprogramming

Metaprogramming is the writing of computer programs that write or manipulate other programs (or themselves) as their data, *or* that do part of the work at compile time that would otherwise be done at runtime. In some cases, this allows programmers to minimize the number of lines of code to express a solution (hence reducing development time), or it gives programs greater flexibility to efficiently handle new situations without recompilation.

The language in which the metaprogram is written is called the metalanguage. The language of the programs that are manipulated is called the *object language*. The ability of a programming language to be its own metalanguage is called *reflection* or *reflexivity*.

Reflection is a valuable language feature to facilitate metaprogramming. Having the programming language itself as a first-class data type (as in Lisp, Forth or Rebol) is also very useful. Generic programming invokes a metaprogramming facility within a language, in those languages supporting it.

Metaprogramming usually works in one of three ways. The first way is to expose the internals of the run-time engine to the programming code through application programming interfaces (APIs). The second approach is dynamic execution of expressions that contain programming commands, often composed from strings, but can also be from other methods using arguments and/or context.^[1] Thus, "programs can write programs." Although both approaches can be used in the same language, most languages tend to lean toward one or the other.

The third way is to step outside the language entirely. General purpose program transformation systems, which accept language descriptions and can carry out arbitrary transformations on those languages, are direct implementations of general metaprogramming. This allows metaprogramming to be applied to virtually any target language without regard to whether that target language has any metaprogramming abilities of its own.

Approaches

In statically typed functional languages

- Usage of dependent types allows proving that generated code is never invalid.^[2]

Template meta-programming

- C "X Macros"
- C++ Templates

Staged meta-programming

- MetaML
- MetaOCaml

Macro systems

- Scheme hygienic macros
- MacroML
- Template Haskell

IBM/360 assembler

The IBM/360 and derivatives had powerful Assembler macro facilities that were often used to generate complete programs or sections of programs (for different operating systems for instance). Macros provided with CICS transaction processing system had Assembler macros that generated COBOL statements as a pre-processing step.

Examples

A simple example of a metaprogram is this bash script, which is an example of generative programming:

```
#!/bin/bash
# metaprogram
echo '#!/bin/bash' >program
for ((I=1; I<=992; I++)) do
    echo "echo $I" >>program
done
chmod +x program
```

This script (or program) generates a new 993-line program that prints out the numbers 1–992. This is only an illustration of how to use code to write more code; it is not the most efficient way to print out a list of numbers. Nonetheless, a programmer can write and execute this metaprogram in less than a minute, and will have generated exactly 1000 lines of code in that amount of time.

A quine is a special kind of metaprogram that produces its own source code as its output.

Not all metaprogramming involves generative programming. If programs are modifiable at runtime or if incremental compilation is available (such as in C#, Forth, Frink, Groovy, JavaScript, Lisp, Lua, Perl, PHP, Python, REBOL, Ruby, Smalltalk, and Tcl), then techniques can be used to perform metaprogramming without actually generating source code.

Lisp is probably the quintessential language with metaprogramming facilities, both because of its historical precedence and because of the simplicity and power of its metaprogramming. In Lisp metaprogramming, the quasiquote operator (typically a comma) introduces code that is evaluated at program definition time rather than at run time. The metaprogramming language is thus identical to the host programming language, and existing Lisp routines can be directly reused for metaprogramming, if desired.

This approach has been implemented in other languages by incorporating an interpreter in the program, which works directly with the program's data. There are implementations of this kind for some common high-level languages, such as RemObject's Pascal Script [3] for Object Pascal.

One style of metaprogramming is to employ domain-specific programming languages (DSLs). A fairly common example of using DSLs involves generative metaprogramming: lex and yacc, two tools used to generate lexical analyzers and parsers, let the user describe the language using regular expressions and context-free grammars, and embed the complex algorithms required to efficiently parse the language.

Implementations

- ASF+SDF Meta Environment
- DMS Software Reengineering Toolkit
- Intentional Programming
- Joose (JavaScript)
- JetBrains MPS
- Moose (Perl)
- Nemerle
- Stratego/XT
- Template Haskell

Notes

- [1] for example, `instance_eval` in Ruby takes a string or an anonymous function. "Rdoc for Class: BasicObject (Ruby 1.9.3) - `instance_eval`" (http://www.ruby-doc.org/core-1.9.3/BasicObject.html#method-i-instance_eval). . Retrieved 30 December 2011.
- [2] Chlipala, Adam (June 2010). "Ur: statically-typed metaprogramming with type-level record computation" (<http://adam.chlipala.net/papers/UrPLDI10/UrPLDI10.pdf>). *ACM SIGPLAN Notices*. PLDI '10 **45** (6): 122–133. doi:10.1145/1809028.1806612. . Retrieved 29 August 2012.
- [3] <http://www.remobjects.com/?ps>

External links

- c2.com Wiki: Metaprogramming article (<http://c2.com/cgi/wiki?MetaProgramming>)
- Meta Programming (<http://www.program-transformation.org/Transform/MetaProgramming>) on the Program Transformation Wiki
- Code generation Vs Metaprogramming (<http://www.qcodo.com/wiki/article/background/metaprogramming>)
- "Solenoid" (<http://solenoid.schematronic.org>): The first metaprogramming framework for eXist-db (<http://exist-db.org>)
- The Art of Enterprise Metaprogramming (<http://www.ibm.com/developerworks/linux/library/l-metaprog3/?ca=dgr-wikiaMetaprogP3>)

Generic programming

In the simplest definition, **generic programming** is a style of computer programming in which algorithms are written in terms of *to-be-specified-later* types that are then *instantiated* when needed for specific types provided as parameters. This approach, pioneered by Ada in 1983, permits writing common functions or types that differ only in the set of types on which they operate when used, thus reducing duplication. Such software entities are known as *generics* in Ada, Eiffel, Java, C#, F#, and Visual Basic .NET; *parametric polymorphism* in ML, Scala and Haskell (the Haskell community also uses the term "generic" for a related but somewhat different concept); *templates* in C++ and D; and *parameterized types* in the influential 1994 book *Design Patterns*. The authors of *Design Patterns* note that this technique, especially when combined with delegation, is very powerful but that "[dynamic], highly parameterized software is harder to understand than more static software."^[1]

The term **generic programming** was originally coined by David Musser and Alexander Stepanov^[2] in a more specific sense than the above, to describe an approach to software decomposition whereby fundamental requirements on types are abstracted from across concrete examples of algorithms and data structures and formalised as concepts, analogously to the abstraction of algebraic theories in abstract algebra.^[3] Early examples of this programming approach were implemented in Scheme and Ada,^[4] although the best known example is the Standard Template Library (STL)^{[5][6]} in which is developed a theory of iterators which is used to decouple sequence data structures and the algorithms operating on them.

For example, given N sequence data structures, e.g. singly linked list, vector etc., and M algorithms to operate on them, e.g. `find`, `sort` etc., a direct approach would implement each algorithm specifically for each data structure, giving NM combinations to implement. However, in the generic programming approach, each data structure returns a model of an iterator concept (a simple value type which can be dereferenced to retrieve the current value, or changed to point to another value in the sequence) and each algorithm is instead written generically with arguments of such iterators, e.g. a pair of iterators pointing to the beginning and end of the subsequence to process. Thus, only $N + M$ data structure-algorithm combinations need be implemented. Several iterator concepts are specified in the STL, each a refinement of more restrictive concepts e.g. forward iterators only provide movement to the next value in a sequence (e.g. suitable for a singly linked list), whereas a random-access iterator also provides direct constant-time access to any element of the sequence (e.g. suitable for a vector). An important point is that a data structure will return a model of the most general concept that can be implemented efficiently—computational complexity requirements are explicitly part of the concept definition. This limits which data structures a given algorithm can be applied to and such complexity requirements are a major determinant of data structure choice. Generic programming similarly has been applied in other domains e.g. graph algorithms.^[7]

Note that although this approach often utilizes language features of compile-time genericity/templates, it is in fact independent of particular language-technical details. Generic programming pioneer Alexander Stepanov wrote: "Generic programming is about abstracting and classifying algorithms and data structures. It gets its inspiration from Knuth and not from type theory. Its goal is the incremental construction of systematic catalogs of useful, efficient and abstract algorithms and data structures. Such an undertaking is still a dream.",^[8] and "I believe that iterator theories are as central to Computer Science as theories of rings or Banach spaces are central to Mathematics."^[9] Following Stepanov, Bjarne Stroustrup defined generic programming without mentioning language features: "[I]ift algorithms and data structures from concrete examples to their most general and abstract form."^[10]

Other programming paradigms that have been described as generic programming includes *datatype generic programming* as described in "Generic Programming — an Introduction".^[11] The *Scrap your boilerplate* approach is a lightweight generic programming approach for Haskell (Lämmel and Peyton Jones, 2003).^[12]

In this article we distinguish the high-level programming paradigms of *generic programming*, above, from the lower-level programming language *genericity mechanisms* used to implement them (see Programming language support for genericity). For further discussion and comparison of generic programming paradigms, see.^[13]

Programming language support for genericity

Genericity facilities have existed in high-level languages since at least the 1970s in languages such as CLU and Ada, and were subsequently adopted by many object-based and object-oriented languages, including BETA, C++, D, Eiffel, Java, and DEC's now defunct Trellis-Owl language. Implementations of generics in languages such as Java and C# are formally based on the notion of parametricity, due to John C. Reynolds.

Genericity is implemented and supported differently in various programming languages; the term "generic" has also been used differently in various programming contexts. For example, in Forth the compiler can execute code while compiling and one can create new *compiler keywords* and new implementations for those words on the fly. It has few *words* that expose the compiler behaviour and therefore naturally offers *genericity* capacities which, however, are not referred to as such in most Forth texts. The term has been used in functional programming, specifically in Haskell-like languages, which use a structural type system where types are always parametric and the actual code on those types is generic. These usages still serve a similar purpose of code-saving and the rendering of an abstraction.

Arrays and structs can be viewed as predefined generic types. Every usage of an array or struct type instantiates a new concrete type, or reuses a previous instantiated type. Array element types and struct element types are parameterized types, which are used to instantiate the corresponding generic type. All this is usually built-in in the compiler and the syntax differs from other generic constructs. Some extensible programming languages try to unify built-in and user defined generic types.

A broad survey of genericity mechanisms in programming languages follows. For a specific survey comparing suitability of mechanisms for generic programming, see.^[14]

In object-oriented languages

When creating container classes in statically typed languages, it is inconvenient to have to write specific implementations for each datatype contained, especially if the code for each datatype is virtually identical. For example, in C++, this duplication of code can be circumvented by defining a template class:

```
template<typename T>
class List
{
    /* class contents */
};

List<Animal> list_of_animals;
List<Car> list_of_cars;
```

Above, `T` is a placeholder for whatever type is specified when the list is created. These "containers-of-type-T", commonly called templates allow a class to be reused with different datatypes as long as certain contracts such as subtypes and signature are kept. This genericity mechanism should not be confused with *inclusion polymorphism*, which is the algorithmic usage of exchangeable sub-classes: for instance, a list of objects of type `Moving_Object` containing objects of type `Animal` and `Car`. Templates can also be used for type-independent functions as in the `Swap` example below:

```
template<typename T>
void Swap(T & a, T & b) // "&" passes parameters by reference
{
    T temp = b;
    b = a;
    a = temp;
}
```

```

string hello = "world!", world = "Hello, ";
Swap( world, hello );
cout << hello << world << endl; //Output is "Hello, world!"

```

The C++ template construct used above is widely cited as the genericity construct that popularized the notion among programmers and language designers and supports many generic programming idioms. The D programming language also offers fully generic-capable templates based on the C++ precedent but with a simplified syntax. The Java programming language has provided genericity facilities syntactically based on C++'s since the introduction of J2SE 5.0.

C# 2.0, Chrome 1.5 and Visual Basic .NET 2005 have constructs that take advantage of the support for generics present in the Microsoft .NET Framework since version 2.0.

Dynamic typing (such as is featured in Objective-C) and judicious use of protocols circumvent the need for use of genericity mechanisms, since there exists a general type to contain any object. While Java does so also, the casting that needs to be done breaks the discipline of static typing, and generics are one way of achieving some of the benefits of dynamic typing with the advantages of having static typing.

Generics in Ada

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.

Example

The specification of a generic package:

```

generic
  Max_Size : Natural; -- a generic formal value
  type Element_Type is private; -- a generic formal type; accepts any
nonlimited type
package Stacks is
  type Size_Type is range 0 .. Max_Size;
  type Stack is limited private;
  procedure Create (S : out Stack;
                    Initial_Size : in Size_Type := Max_Size);
  procedure Push (Into : in out Stack; Element : in Element_Type);
  procedure Pop (From : in out Stack; Element : out Element_Type);
  Overflow : exception;
  Underflow : exception;
private
  subtype Index_Type is Size_Type range 1 .. Max_Size;
  type Vector is array (Index_Type range <>) of Element_Type;

```

```

type Stack (Allocated_Size : Size_Type := 0) is record
    Top : Index_Type;
    Storage : Vector (1 .. Allocated_Size);
end record;
end Stacks;

```

Instantiating the generic package:

```

type Bookmark_Type is new Natural;
-- records a location in the text document we are editing

package Bookmark_Stacks is new Stacks (Max_Size => 20,
                                         Element_Type => Bookmark_Type);
-- Allows the user to jump between recorded locations in a document

```

Using an instance of a generic package:

```

type Document_Type is record
    Contents : Ada.Strings.Unbounded.Unbounded_String;
    Bookmarks : Bookmark_Stacks.Stack;
end record;

procedure Edit (Document_Name : in String) is
    Document : Document_Type;
begin
    -- Initialise the stack of bookmarks:
    Bookmark_Stacks{{Not a typol.}}Create (S => Document{{Not a
typol.}}Bookmarks, Initial_Size => 10);
    -- Now, open the file Document_Name and read it in...
end Edit;

```

Advantages and limitations

The language syntax allows precise specification of constraints on generic formal parameters. For example, it is possible to specify that a generic formal type will only accept a modular type as the actual. It is also possible to express constraints *between* generic formal parameters; for example:

```

generic
    type Index_Type is (<>); -- must be a discrete type
    type Element_Type is private; -- can be any nonlimited type
    type Array_Type is array (Index_Type range <>) of Element_Type;

```

In this example, Array_Type is constrained by both Index_Type and Element_Type. When instantiating the unit, the programmer must pass an actual array type that satisfies these constraints.

The disadvantage of this fine-grained control is a complicated syntax, but, because all generic formal parameters are completely defined in the specification, the compiler can instantiate generics without looking at the body of the generic.

Unlike C++, Ada does not allow specialised generic instances, and requires that all generics be instantiated explicitly. These rules have several consequences:

- the compiler can implement *shared generics*: the object code for a generic unit can be shared between all instances (unless the programmer requests inlining of subprograms, of course). As further consequences:

- there is no possibility of code bloat (code bloat is common in C++ and requires special care, as explained below).
- it is possible to instantiate generics at run-time, as well as at compile time, since no new object code is required for a new instance.
- actual objects corresponding to a generic formal object are always considered to be nonstatic inside the generic; see Generic formal objects in the Wikibook for details and consequences.
- all instances of a generic being exactly the same, it is easier to review and understand programs written by others; there are no "special cases" to take into account.
- all instantiations being explicit, there are no hidden instantiations that might make it difficult to understand the program.
- Ada does not permit "template metaprogramming", because it does not allow specialisations.

Templates in C++

C++ uses templates to enable generic programming techniques. The C++ Standard Library includes the Standard Template Library or STL that provides a framework of templates for common data structures and algorithms. Templates in C++ may also be used for template metaprogramming, which is a way of pre-evaluating some of the code at compile-time rather than run-time. Using template specialization, C++ Templates are considered Turing complete.

Technical overview

There are two kinds of templates: function templates and class templates. A *function template* is a pattern for creating ordinary functions based upon the parameterizing types supplied when instantiated. For example, the C++ Standard Template Library contains the function template `max(x, y)` which creates functions that return either `x` or `y`, whichever is larger. `max()` could be defined like this:

```
template <typename T>
T max(T x, T y)
{
    return x < y ? y : x;
}
```

Specializations of this function template, instantiations with specific types, can be called just like an ordinary function:

```
cout << max(3, 7); // outputs 7
```

The compiler examines the arguments used to call `max` and determines that this is a call to `max(int, int)`. It then instantiates a version of the function where the parameterizing type `T` is `int`, making the equivalent of the following function:

```
int max(int x, int y)
{
    return x < y ? y : x;
}
```

This works whether the arguments `x` and `y` are integers, strings, or any other type for which the expression `x < y` is sensible, or more specifically, for any type for which `operator<` is defined. Common inheritance is not needed for the set of types that can be used, and so it is very similar to duck typing. A program defining a custom data type can use operator overloading to define the meaning of `<` for that type, thus allowing its use with the `max()` function template. While this may seem a minor benefit in this isolated example, in the context of a

comprehensive library like the STL it allows the programmer to get extensive functionality for a new data type, just by defining a few operators for it. Merely defining `<` allows a type to be used with the standard `sort()`, `stable_sort()`, and `binary_search()` algorithms or to be put inside data structures such as `sets`, `heaps`, and associative arrays.

C++ templates are completely type safe at compile time. As a demonstration, the standard type `complex` does not define the `<` operator, because there is no strict order on complex numbers. Therefore `max(x, y)` will fail with a compile error if `x` and `y` are `complex` values. Likewise, other templates that rely on `<` cannot be applied to `complex` data unless a comparison (in the form of a functor or function) is provided. E.g.: A `complex` cannot be used as key for a `map` unless a comparison is provided. Unfortunately, compilers historically generate somewhat esoteric, long, and unhelpful error messages for this sort of error. Ensuring that a certain object adheres to a method protocol can alleviate this issue. Languages which use `compare` instead of `<` can also use `complex` values as keys.

The second kind of template, a *class template*, extends the same concept to classes. A class template specialization is a class. Class templates are often used to make generic containers. For example, the STL has a linked list container. To make a linked list of integers, one writes `list<int>`. A list of strings is denoted `list<string>`. A list has a set of standard functions associated with it, which work for any compatible parameterizing types.

Template specialization

A powerful feature of C++'s templates is *template specialization*. This allows alternative implementations to be provided based on certain characteristics of the parameterized type that is being instantiated. Template specialization has two purposes: to allow certain forms of optimization, and to reduce code bloat.

For example, consider a `sort()` template function. One of the primary activities that such a function does is to swap or exchange the values in two of the container's positions. If the values are large (in terms of the number of bytes it takes to store each of them), then it is often quicker to first build a separate list of pointers to the objects, sort those pointers, and then build the final sorted sequence. If the values are quite small however it is usually fastest to just swap the values in-place as needed. Furthermore if the parameterized type is already of some pointer-type, then there is no need to build a separate pointer array. Template specialization allows the template creator to write different implementations and to specify the characteristics that the parameterized type(s) must have for each implementation to be used.

Unlike function templates, class templates can be partially specialized. That means that an alternate version of the class template code can be provided when some of the template parameters are known, while leaving other template parameters generic. This can be used, for example, to create a default implementation (the *primary specialization*) that assumes that copying a parameterizing type is expensive and then create partial specializations for types that are cheap to copy, thus increasing overall efficiency. Clients of such a class template just use specializations of it without needing to know whether the compiler used the primary specialization or some partial specialization in each case. Class templates can also be *fully specialized*, which means that an alternate implementation can be provided when all of the parameterizing types are known.

Advantages and disadvantages

Some uses of templates, such as the `max()` function, were previously filled by function-like preprocessor macros (a legacy of the C programming language). For example, here is a possible `max()` macro:

```
#define max(a,b) ((a) < (b) ? (b) : (a))
```

Both macros and templates are expanded at compile time. Macros are always expanded inline; templates can also be expanded as inline functions when the compiler deems it appropriate. Thus both function-like macros and function templates have no run-time overhead.

However, templates are generally considered an improvement over macros for these purposes. Templates are type-safe. Templates avoid some of the common errors found in code that makes heavy use of function-like macros, such as evaluating parameters with side effects twice. Perhaps most importantly, templates were designed to be applicable to much larger problems than macros.

There are three primary drawbacks to the use of templates: compiler support, poor error messages, and code bloat. Many compilers historically have poor support for templates, thus the use of templates can make code somewhat less portable. Support may also be poor when a C++ compiler is being used with a linker which is not C++-aware, or when attempting to use templates across shared library boundaries. Most modern compilers however now have fairly robust and standard template support, and the new C++ standard, C++11, further address these issues.

Almost all compilers produce confusing, long, or sometimes unhelpful error messages when errors are detected in code that uses templates.^[15] This can make templates difficult to develop.

Finally, the use of templates requires the compiler to generate a separate *instance* of the templated class or function for every permutation of type parameters used with it. (This is necessary because types in C++ are not all the same size, and the sizes of data fields are important to how classes work.) So the indiscriminate use of templates can lead to code bloat, resulting in excessively large executables. However, judicious use of template specialization can dramatically reduce such code bloat in some cases. The extra instantiations generated by templates can also cause debuggers to have difficulty working gracefully with templates. For example, setting a debug breakpoint within a template from a source file may either miss setting the breakpoint in the actual instantiation desired or may set a breakpoint in every place the template is instantiated.

Also, because the compiler needs to perform macro-like expansions of templates and generate different instances of them at compile time, the implementation source code for the templated class or function must be available (e.g. included in a header) to the code using it. Tempered classes or functions, including much of the Standard Template Library (STL), cannot be compiled. (This is in contrast to non-tempered code, which may be compiled to binary, providing only a declarations header file for code using it.) This may be a disadvantage by exposing the implementing code, which removes some abstractions, and could restrict its use in closed-source projects.

Genericity in Eiffel

Generic classes have been a part of Eiffel since the original method and language design. The foundation publications of Eiffel,^{[16][17]} use the term *genericity* to describe the creation and use of generic classes.

Basic/Unconstrained genericity

Generic classes are declared with their class name and a list of one or more *formal generic parameters*. In the following code, class `LIST` has one formal generic parameter `G`

```
class
  LIST [G]
  ...
feature -- Access
  item: G
```

```

    -- The item currently pointed to by cursor
    ...
feature  -- Element change
    put (new_item: G)
        -- Add `new_item' at the end of the list
    ...

```

The formal generic parameters are placeholders for arbitrary class names which will be supplied when a declaration of the generic class is made, as shown in the two *generic derivations* below, where ACCOUNT and DEPOSIT are other class names. ACCOUNT and DEPOSIT are considered *actual generic parameters* as they provide real class names to substitute for G in actual use.

```

list_of_accounts: LIST [ACCOUNT]
    -- Account list

list_of_deposits: LIST [DEPOSIT]
    -- Deposit list

```

Within the Eiffel type system, although class LIST [G] is considered a class, it is not considered a type. However, a generic derivation of LIST [G] such as LIST [ACCOUNT] is considered a type.

Constrained genericity

For the list class shown above, an actual generic parameter substituting for G can be any other available class. To constrain the set of classes from which valid actual generic parameters can be chosen, a *generic constraint* can be specified. In the declaration of class SORTED_LIST below, the generic constraint dictates that any valid actual generic parameter will be a class which inherits from class COMPARABLE. The generic constraint ensures that elements of a SORTED_LIST can in fact be sorted.

```

class
    SORTED_LIST [G -> COMPARABLE]

```

Generics in Java

Support for the *generics*, or "containers-of-type-T" were added to the Java programming language in 2004 as part of J2SE 5.0. In Java, generics are only checked at compile time for type correctness. The generic type information is then removed via a process called type erasure, to maintain compatibility with old JVM implementations, making it unavailable at runtime. For example, a `List<String>` is converted to the raw type `List`. The compiler inserts type casts to convert the elements to the `String` type when they are retrieved from the list, reducing performance compared to other implementations such as C++ templates.

Genericity in .NET

Generics were added as part of .NET Framework 2.0 in November 2005, based on a research prototype from Microsoft Research started in 1999.^[18] Although similar to generics in Java, .NET generics do not apply type erasure, but implement generics as a first class mechanism in the runtime using reification. This design choice provides additional functionality, such as allowing reflection with preservation of generic types, as well as alleviating some of the limitations of erasure (such as being unable to create generic arrays).^{[19][20]} This also means that there is no performance hit from runtime casts and normally expensive boxing conversions. When primitive and value types are used as generic arguments, they get specialized implementations, allowing for efficient generic collections and methods. As in C++ and Java, nested generic types such as `Dictionary<string, List<int>>` are valid types, however are advised against for member signatures in code analysis design rules.^[21]

.NET allows six varieties of generic type constraints using the `where` keyword including restricting generic types to be value types, to be classes, to have constructors, and to inherit from interfaces.^[22] Below is an example with an interface constraint:

```
using System;

class Sample {
    static void Main() {
        int[] array = { 0, 1, 2, 3 };
        MakeAtLeast<int>(array, 2); // Change array to { 2, 2, 2, 3 }
        foreach (int i in array)
            Console.WriteLine(i); // Print results.
        Console.ReadKey(true);
    }

    static void MakeAtLeast<T>(T[] list, T lowest) where T : IComparable<T> {
        for (int i = 0; i < list.Length; i++)
            if (list[i].CompareTo(lowest) < 0)
                list[i] = lowest;
    }
}
```

The `MakeAtLeast()` method allows operation on arrays, with elements of generic type `T`. The method's type constraint indicates that the method is applicable to any type `T` that implements the generic `IComparable<T>` interface. This ensures a compile time error if the method is called if the type does not support comparison. The interface provides the generic method `CompareTo(T)`.

The above method could also be written without generic types, simply using the non-generic `Array` type. However since arrays are contravariant, the casting would not be type safe, and compiler may miss errors that would otherwise be caught while making use of the generic types. In addition, the method would need to access the array items as objects instead, and would require casting to compare two elements. (For value types like types such as `int` this requires a boxing conversion, although this can be worked around using the `Comparer<T>` class, as is done in the standard collection classes.)

A notable behavior of static members in a generic .NET class is static member instantiation per run-time type (see example below).

```
//A generic class
public class GenTest<T>
{
    //A static variable - will be created for each type on
    //refraction
    static CountedInstances OnePerType = new CountedInstances();

    //a data member
    private T mT;

    //simple constructor
    public GenTest(T pT)
    {
```

```

        mT = pT;
    }

}

//a class
public class CountedInstances
{
    //Static variable - this will be incremented once per instance
    public static int Counter;

    //simple constructor
    public CountedInstances()
    {
        //increase counter by one during object instantiation
        CountedInstances.Counter++;
    }
}

//main code entry point
//at the end of execution, CountedInstances{{Not a typo/.}}Counter =
2
GenTest<int> g1 = new GenTest<int>(1);
GenTest<int> g11 = new GenTest<int>(11);
GenTest<int> g111 = new GenTest<int>(111);
GenTest<double> g2 = new GenTest<double>(1.0);

```

Genericity in Delphi

Delphi's Object Pascal dialect acquired generics in the Delphi 2007 release, initially only with the (now discontinued) .NET compiler before being added to the native code one in the Delphi 2009 release. The semantics and capabilities of Delphi generics are largely modelled on those had by generics in .NET 2.0, though the implementation is by necessity quite different. Here's a more or less direct translation of the first C# example shown above:

```

program Sample;

{$APPTYPE CONSOLE}

uses
  Generics.Defaults; //for IComparer<>

type
  TUtils = class
    class procedure MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T;
      Comparer: IComparer<T>); overload;
    class procedure MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T);
      overload;
  end;

```

```

class procedure TUtils.MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T;
  Comparer: IComparer<T>);

var
  I: Integer;
begin
  if Comparer = nil then Comparer := TComparer<T>.Default;
  for I := Low(Arr) to High(Arr) do
    if Comparer.Compare(Arr[I], Lowest) < 0 then
      Arr[I] := Lowest;
end;

class procedure TUtils.MakeAtLeast<T>(Arr: TArray<T>; const Lowest: T);
begin
  MakeAtLeast<T>(Arr, Lowest, nil);
end;

var
  Ints: TArray<Integer>;
  Value: Integer;
begin
  Ints := TArray<Integer>.Create(0, 1, 2, 3);
  TUtils.MakeAtLeast<Integer>(Ints, 2);
  for Value in Ints do
    WriteLn(Value);
  ReadLn;
end.

```

As with C#, methods as well as whole types can have one or more type parameters. In the example, TArray is a generic type (defined by the language) and MakeAtLeast a generic method. The available constraints are very similar to the available constraints in C#: any value type, any class, a specific class or interface, and a class with a parameterless constructor. Multiple constraints act as an additive union.

Genericity in Free Pascal

Free Pascal implemented generics before Delphi, and with different syntax and semantics. However, work is now underway to implement Delphi generics alongside native FPC ones (see Wiki ^[23]). This allows Free Pascal programmers to use generics in whatever style they prefer.

Delphi and Free Pascal example:

```

// Delphi style

unit A;

{$ifdef fpc}
{$mode delphi}
{$endif}

interface

type

```

```
TGenericClass<T> = class
  function Double(const AValue: T): T;
end;

implementation

function TGenericClass<T>.Double(const AValue: T): T;
begin
  Result := AValue + AValue;
end;

end.

// Free Pascal's ObjFPC style
unit B;

{$ifdef fpc}
{$mode objfpc}
{$endif}

interface

type
  generic TGenericClass<T> = class
    function Double(const AValue: T): T;
  end;

implementation

function TGenericClass.Double(const AValue: T): T;
begin
  Result := AValue + AValue;
end;

end.

// example usage, Delphi style
program TestGenDelphi;

{$ifdef fpc}
{$mode delphi}
{$endif}

uses
  A, B;

var
```

```

GC1: A.TGenericClass<Integer>;
GC2: B.TGenericClass<String>;
begin
  GC1 := A.TGenericClass<Integer>.Create;
  GC2 := B.TGenericClass<String>.Create;
  WriteLn(GC1.Double(100)); // 200
  WriteLn(GC2.Double('hello')); // hellohello
  GC1.Free;
  GC2.Free;
end.

// example usage, ObjFPC style
program TestGenDelphi;

{$ifdef fpc}
{$mode objfpc}
{$endif}

uses
  A, B;

// required in ObjFPC
type
  TAGenericClassInt = specialize A.TGenericClass<Integer>;
  TBGenericClassString = specialize B.TGenericClass<String>;
var
  GC1: TAGenericClassInt;
  GC2: TBGenericClassString;
begin
  GC1 := TAGenericClassInt.Create;
  GC2 := TBGenericClassString.Create;
  WriteLn(GC1.Double(100)); // 200
  WriteLn(GC2.Double('hello')); // hellohello
  GC1.Free;
  GC2.Free;
end.

```

Functional languages

Genericity in Haskell

The type class mechanism of Haskell supports generic programming. Six of the predefined type classes in Haskell (including `Eq`, the types that can be compared for equality, and `Show`, the types whose values can be rendered as strings) have the special property of supporting *derived instances*. This means that a programmer defining a new type can state that this type is to be an instance of one of these special type classes, without providing implementations of the class methods as is usually necessary when declaring class instances. All the necessary methods will be "derived" – that is, constructed automatically – based on the structure of the type. For instance, the following declaration of a type of binary trees states that it is to be an instance of the classes `Eq` and `Show`:

```
data BinTree a = Leaf a | Node (BinTree a) a (BinTree a)
    deriving (Eq, Show)
```

This results in an equality function (`==`) and a string representation function (`show`) being automatically defined for any type of the form `BinTree T` provided that `T` itself supports those operations.

The support for derived instances of `Eq` and `Show` makes their methods `==` and `show` generic in a qualitatively different way from parametrically polymorphic functions: these "functions" (more accurately, type-indexed families of functions) can be applied to values of various types, and although they behave differently for every argument type, little work is needed to add support for a new type. Ralf Hinze (2004) has shown that a similar effect can be achieved for user-defined type classes by certain programming techniques. Other researchers have proposed approaches to this and other kinds of genericity in the context of Haskell and extensions to Haskell (discussed below).

PolyP

PolyP was the first generic programming language extension to Haskell. In PolyP, generic functions are called *polytypic*. The language introduces a special construct in which such polytypic functions can be defined via structural induction over the structure of the pattern functor of a regular datatype. Regular datatypes in PolyP are a subset of Haskell datatypes. A regular datatype `t` must be of kind $* \rightarrow *$, and if `a` is the formal type argument in the definition, then all recursive calls to `t` must have the form `t a`. These restrictions rule out higher-kinded datatypes as well as nested datatypes, where the recursive calls are of a different form. The `flatten` function in PolyP is here provided as an example:

```
flatten :: Regular d => d a -> [a]
flatten = cata f1

polytypic f1 :: f a [a] -> [a]
  case f of
    g+h -> either f1 f1
    g*h -> \ (x,y) -> f1 x ++ f1 y
    () -> \x -> []
    Par -> \x -> [x]
    Rec -> \x -> x
    d@g -> concat . flatten . pmap f1
    Con t -> \x -> []

cata :: Regular d => (FunctorOf d a b -> b) -> d a -> b
```

Generic Haskell

Generic Haskell is another extension to Haskell, developed at Utrecht University in the Netherlands. The extensions it provides are:

- *Type-indexed values* are defined as a value indexed over the various Haskell type constructors (unit, primitive types, sums, products, and user-defined type constructors). In addition, we can also specify the behaviour of a type-indexed values for a specific constructor using *constructor cases*, and reuse one generic definition in another using *default cases*.

The resulting type-indexed value can be specialised to any type.

- *Kind-indexed types* are types indexed over kinds, defined by giving a case for both $*$ and $k \rightarrow k'$. Instances are obtained by applying the kind-indexed type to a kind.

- Generic definitions can be used by applying them to a type or kind. This is called *generic application*. The result is a type or value, depending on which sort of generic definition is applied.
- *Generic abstraction* enables generic definitions be defined by abstracting a type parameter (of a given kind).
- *Type-indexed types* are types which are indexed over the type constructors. These can be used to give types to more involved generic values. The resulting type-indexed types can be specialised to any type.

As an example, the equality function in Generic Haskell:^[24]

```

type Eq {[ * ]} t1 t2 = t1 -> t2 -> Bool
type Eq {[ k -> l ]} t1 t2 = forall u1 u2. Eq {[ k ]} u1 u2 -> Eq {[ l
]} (t1 u1) (t2 u2)

eq { | t :: k | } :: Eq {[ k ]} t t
eq { | Unit | } _ _ = True
eq { | :+: | } eqA eqB (Inl a1) (Inl a2) = eqA a1 a2
eq { | :+: | } eqA eqB (Inr b1) (Inr b2) = eqB b1 b2
eq { | :+: | } eqA eqB _ _ = False
eq { | :*: | } eqA eqB (a1 :*: b1) (a2 :*: b2) = eqA a1 a2 && eqB b1

b2
eq { | Int | } = (==)
eq { | Char | } = (==)
eq { | Bool | } = (==)

```

Clean

Clean offers generic programming based PolyP and the generic Haskell as supported by the GHC>=6.0. It parametrizes by kind as those but offers overloading.

Other languages

The ML family of programming languages support generic programming through parametric polymorphism and generic modules called *functors*. Both Standard ML and OCaml provide functors, which are similar to class templates and to Ada's generic packages. Scheme syntactic abstractions also have a connection to genericity – these are in fact a superset of templating à la C++.

Notes

- [1] Gamma, et al. 1994, p. 21.
- [2] David R. Musser and Alexander A. Stepanov: "Generic Programming" in P. Gianni, ed., Symbolic and Algebraic Computation: International symposium ISSAC 1988, pages 13-25
- [3] Alexander Stepanov and Paul McJones (June 19, 2009). *Elements of Programming*. Addison-Wesley Professional. ISBN 978-0-321-63537-2.
- [4] David R. Musser and Alexander A. Stepanov: A library of generic algorithms in Ada. Proceedings of the 1987 annual ACM SIGAda international conference on Ada, pages 216-225
- [5] Alexander Stepanov and Meng Lee: The Standard Template Library. HP Laboratories Technical Report 95-11(R.1), 14 November 1995
- [6] Matthew H. Austern: Generic programming and the STL: using and extending the C++ Standard Template Library. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA 1998
- [7] Jeremy G. Siek, Lie-Quan Lee, Andrew Lumsdaine: The Boost Graph Library: User Guide and Reference Manual. Addison-Wesley 2001
- [8] Bjarne 2007, p. 17.
- [9] Lo Russo, Graziano. "An Interview with A. Stepanov" (<http://www.stlport.org/resources/StepanovUSA.html>). .
- [10] Bjarne 2007, p. 35.
- [11] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens (1999). *Advanced Functional Programming, volume 1608 of Lecture Notes in Computer Science, chapter Generic Programming — An introduction, pages 28–115*. Verlag.
- [12] *Uniplate* is a package with a similar basic approach (<http://www-users.cs.york.ac.uk/~ndm/uniplate/>)

- [13] Gabriel Dos Reis and Jaakko Järvi (2005). "What is Generic Programming? (preprint LCSD'05)" (http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf) .
- [14] R. Garcia, J. Järvi, A. Lumsdaine, J. Siek, J. Willcock (2005). "An extended comparative study of language support for generic programming (preprint)" (http://www.osl.iu.edu/publications/prints/2005/garcia05:_extended_comparing05.pdf) .
- [15] Stroustrup, Dos Reis (2003): Concepts - Design choices for template argument checking (<http://www.research.att.com/~bs/N1522-concept-criteria.pdf>)
- [16] *Object-Oriented Software Construction*, Prentice Hall, 1988, and *Object-Oriented Software Construction, second edition*, Prentice Hall, 1997.
- [17] *Eiffel: The Language*, Prentice Hall, 1991.
- [18] .NET/C# Generics History: Some Photos From Feb 1999 (<http://blogs.msdn.com/b/dsyme/archive/2011/03/15/net-c-generics-history-some-photos-from-feb-1999.aspx>)
- [19] C#: Yesterday, Today, and Tomorrow: An Interview with Anders Hejlsberg (<http://www.ondotnet.com/pub/a/dotnet/2005/10/17/interview-with-anders-hejlsberg.html>)
- [20] Generics in C#, Java, and C++ (<http://www.artima.com/intv/generics2.html>)
- [21] Code Analysis CA1006: Do not nest generic types in member signatures (<http://msdn.microsoft.com/en-us/library/ms182144.aspx>)
- [22] Constraints on Type Parameters (C# Programming Guide) (<http://msdn2.microsoft.com/en-us/library/d5x73970.aspx>)
- [23] http://wiki.lazarus.freepascal.org/User_Changes_2.6.0
- [24] The Generic Haskell User's Guide (<http://www.cs.uu.nl/research/projects/generic-haskell/compiler/diamond/GHUsersGuide.pdf>)

References

- Stroustrup, Bjarne (2007). "Evolving a language in and for the real world: C++ 1991-2006" (<http://www.research.att.com/~bs/hopl-almost-final.pdf>). ACM HOPL 2007.
- Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2.

Further reading

- Gabriel Dos Reis and Jaakko Järvi, *What is Generic Programming?* (http://lcsd05.cs.tamu.edu/papers/dos_reis_et_al.pdf), LCSD 2005 (<http://lcsd05.cs.tamu.edu>).
- Gibbons, Jeremy (2007). "Datatype-generic programming". In Backhouse, R.; Gibbons, J.; Hinze, R. et al.. Lecture Notes in Computer Science. **4719**. Spring School on Datatype-Generic Programming 2006. Heidelberg: Springer. pp. 1–71. CiteSeerX: 10.1.1.159.1228 (<http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.159.1228>).
- Bertrand Meyer. "Genericity vs Inheritance (<http://se.ethz.ch/~meyer/publications/acm/geninh.pdf>).". In *OOPSLA (First ACM Conference on Object-Oriented Programming Systems, Languages and Applications)*, Portland (Oregon), 29 September–2 October 1986, pages 391–405.

External links

- [generic-programming.org](http://www.generic-programming.org) (<http://www.generic-programming.org>)
- Alexander A. Stepanov, Collected Papers of Alexander A. Stepanov (<http://www.stepanovpapers.com/>) (creator of the STL)

C++/D

- Walter Bright, *Templates Revisited* (<http://www.digitalmars.com/d/templates-revisited.html>).
- David Vandevoorde, Nicolai M Josuttis, *C++ Templates: The Complete Guide*, 2003 Addison-Wesley. ISBN 0-201-73484-2

C#/.NET

- Jason Clark, "Introducing Generics in the Microsoft CLR (<http://msdn.microsoft.com/msdnmag/issues/03/09/.NET/>)," September 2003, *MSDN Magazine*, Microsoft.

- Jason Clark, " More on Generics in the Microsoft CLR (<http://msdn.microsoft.com/msdnmag/issues/03/10/NET/>)," October 2003, *MSDN Magazine*, Microsoft.
- M. Aamir Maniar, Generics.Net (<http://codeplex.com/Wiki/View.aspx?ProjectName=genericsnet>). An open source generics library for C#.

Delphi/Object Pascal

- Nick Hodges, " Delphi 2009 Reviewers Guide (<http://dn.codegear.com/article/38757>)," October 2008, *CodeGear Developer Network*, CodeGear.
- Craig Stuntz, " Delphi 2009 Generics and Type Constraints (<http://blogs.teamb.com/craigstuntz/2008/08/29/37832/>)," October 2008
- Dr. Bob, " Delphi 2009 Generics (<http://www.drbob42.com/examines/examinA4.htm>)"
- Free Pascal: Free Pascal Reference guide Chapter 8: Generics (<http://www.freepascal.org/docs-html/ref/refch8.html>), Michaël Van Canneyt, 2007
- Delphi for Win32: Generics with Delphi 2009 Win32 (<http://sjrd.developpez.com/delphi/tutoriel/generics/>), Sébastien DOERAENE, 2008
- Delphi for .NET: Delphi Generics (http://www.felix-colibri.com/papers/oop_components/delphi_generics_tutorial/delphi_generics_tutorial.html), Felix COLIBRI, 2008

Eiffel

- Eiffel ISO/ECMA specification document (<http://www.ecma-international.org/publications/standards/Ecma-367.htm>)

Haskell

- Johan Jeuring, Sean Leather, José Pedro Magalhães, and Alexey Rodriguez Yakushev. *Libraries for Generic Programming in Haskell* (<http://www.cs.uu.nl/wiki/pub/GP/CourseLiterature/afp08.pdf>). Utrecht University.
- Daev Clarke, Johan Jeuring and Andres Löh, The Generic Haskell user's guide (<http://www.cs.uu.nl/research/projects/generic-haskell/compiler/diamond/GHUsersGuide.pdf>)
- Ralf Hinze, " Generics for the Masses (<http://www.informatik.uni-bonn.de/~ralf/publications/ICFP04.pdf>)," In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP)*, 2004.
- Simon Peyton Jones, editor, *The Haskell 98 Language Report* (<http://haskell.org/onlinereport/index.html>), Revised 2002.
- Ralf Lämmel and Simon Peyton Jones, "Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming," In *Proceedings of the ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI'03)*, 2003. (Also see the website devoted to this research (<http://www.cs.vu.nl/boilerplate/>))
- Andres Löh, *Exploring Generic Haskell* (<http://www.cs.uu.nl/~andres/ExploringGH.pdf>), Ph.D. thesis, 2004 Utrecht University. ISBN 90-393-3765-9
- Generic Haskell: a language for generic programming (<http://www.generic-haskell.org/>)

Java

- Gilad Bracha, *Generics in the Java Programming Language* (<http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf>), 2004.
- Maurice Naftalin and Philip Wadler, *Java Generics and Collections*, 2006, O'Reilly Media, Inc. ISBN 0-596-52775-6
- Peter Sestoft, *Java Precisely, Second Edition*, 2005 MIT Press. ISBN 0-262-69325-9
- Java SE 7 (<http://download.oracle.com/javase/7/docs/>), 2004 Sun Microsystems, Inc.
- Angelika Langer, Java Generics FAQs (<http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ.html>)

Ada (programming language)

Ada

Paradigm(s)	Multi-paradigm
Appeared in	1980
Designed by	<ul style="list-style-type: none"> MIL-STD-1815/Ada 83: Jean Ichbiah Ada 95: Tucker Taft Ada 2005: Tucker Taft
Stable release	Ada 2005 (2007)
Preview release	Ada 2012 ^[1] (November 2011)
Typing discipline	static, strong, safe, nominative
Major implementations	AdaCore GNAT, Green Hills Software Optimising Ada 95 compiler, DDC-I Score
Dialects	SPARK, Ravenscar profile
Influenced by	ALGOL 68, Pascal, C++ (Ada 95), Smalltalk (Ada 95), Java (Ada 2005)
Influenced	C++, Eiffel, PL/SQL, VHDL, Ruby, Java, Seed7
OS	Cross-platform (multi-platform)
Usual filename extensions	.adb .ads
Website	http://www.adaic.org/
 Ada Programming at Wikibooks	

Ada is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages. It has strong built-in language support for explicit concurrency, offering tasks, synchronous message passing, protected objects, and non-determinism. Ada is an international standard; the current version (known as Ada 2005) is defined by joint ISO/ANSI standard,^[2] combined with major Amendment ISO/IEC 8652:1995/Amd 1:2007.^[3]

Ada was originally designed by a team led by Jean Ichbiah of CII Honeywell Bull under contract to the United States Department of Defense (DoD) from 1977 to 1983 to supersede the hundreds of programming languages then used by the DoD. Ada was named after Ada Lovelace (1815–1852), who is sometimes credited as being the first computer programmer.^[4]

Features

Ada was originally targeted at 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).

Notable features of Ada include: strong typing, modularity 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 is simple, consistent, and readable – it 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 mathematical symbols (i.e.: "+", "-", "*" and "/") for basic mathematical operations but avoids using other symbols. Code blocks are delimited by words such as "declare", "begin", and "end", whereas 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 development of 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 detection of 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, railways, banking, military and space technology.^{[5][6]}

Ada's dynamic memory management is high-level and type-safe. Ada does not have generic (and vague) "pointers"; nor does it implicitly declare any pointer type. Instead, all dynamic memory allocation and deallocation must take place through 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 storage 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.

Ada was designed to use the English language standard for comments: the em-dash, as a double-dash ("--") to denote comment text. Comments stop at end of line, so there is no danger of unclosed comments accidentally voiding whole sections of source code. Comments can be nested: prefixing each line (or column) with "--" will skip all that code, while being clearly denoted as a column of repeated "--" down the page. There is no limit to the nesting of comments, thereby allowing prior code, with commented-out sections, to be commented-out as even larger sections. All Unicode characters are allowed in comments, such as for symbolic formulas ($E[0]=m\times c^2$). To the compiler, the double-dash is treated as end-of-line, allowing continued parsing of the language as a context-free grammar.

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. This allows for a better quality of error messages.

Code for complex systems is typically maintained for many years, by programmers other than the original author. These language design principles apply to most software projects, and most phases of software development, but when applied to complex, safety critical projects, benefits in correctness, reliability, and maintainability take precedence over (arguable) costs in initial development.

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 and not just 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 **GPS**, the GNAT Programming Studio.

History

In the 1970s, the US Department of Defense (DoD) was 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 requirements. The result was 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^[7]), 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 programming 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,^[8] but subsequently seemed to recant in the foreword he wrote for an Ada textbook.^[9]

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 just defense-related work. Ichbiah publicly stated that within ten years, only two programming languages would remain, Ada and Lisp.^[10] 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.^[10]

The first validated Ada implementation was the NYU Ada/Ed translator,^[11] certified on April 11, 1983. NYU Ada/Ed is implemented in the high-level set language SETL.^[12]

In 1987, the US Department of Defense began to require the use of Ada (the *Ada mandate*) for every software project where new code was more than 30% of result, though exceptions to this rule were often granted.

By the late 1980s and early 1990s, Ada compilers had improved in performance, but there were still barriers to full exploitation of Ada's abilities, including a tasking model that was different from what most real-time programmers were used to.^[10]

The Department of Defense Ada mandate was effectively removed in 1997, as the DoD began to embrace COTS (commercial off-the-shelf) technology. Similar requirements existed in other NATO countries.

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. aviation and air traffic control, commercial rockets (e.g. Ariane 4 and 5), satellites and other space systems, railway transport and banking.^[6] For example, the fly-by-wire system software in the Boeing 777 was written in Ada. 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^[13]. 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.^{[6][14]}



Augusta Ada King, Countess of Lovelace.

Standardization

The language became an ANSI standard in 1983 (ANSI/MIL-STD 1815A^[15]), and without any further changes 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^[16]) 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 programming language. A Technical Corrigendum to Ada 95 was published in October 2001, and a major Amendment, ISO/IEC 8652:1995/Amd 1:2007^[17], the current version of the standard, 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 programming language and the submission of the reference manual to the International Organization for Standardization (ISO) for approval. ISO/IEC 8652:201z Ed. 3^[18]

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 or inspired from Pascal.

"Hello, world!" in Ada

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

```
with Ada.Text_IO; use Ada.Text_IO;
procedure Hello is
begin
  Put_Line ("Hello, world!");
end Hello;
```

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

```
gnatmake hello.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:

```
type Day_type is range 1 .. 31;
type Month_type is range 1 .. 12;
type Year_type is range 1800 .. 2100;
type Hours is mod 24;
type Weekday is (Monday, Tuesday, Wednesday, Thursday, Friday,
Saturday, Sunday);

type Date is
  record
    Day    : Day_type;
    Month : Month_type;
    Year  : Year_type;
  end record;
```

Types can be refined by declaring subtypes:

```
subtype Working_Hours is Hours range 0..12;           -- at most 12
Hours to work a day
subtype Working_Day is Weekday range Monday .. Friday; -- Days to
```

```

work

Work_Load: constant array(Working_Day) of Working_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.^[19] Ada 95 adds additional 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 /= b loop
  Ada.Text_IO.Put_Line ("Waiting");
end loop;

if a > b then
  Ada.Text_IO.Put_Line ("Condition met");
else
  Ada.Text_IO.Put_Line ("Condition not met");
end if;

for i in 1 .. 10 loop
  Ada.Text_IO.Put ("Iteration: ");
  Ada.Text_IO.Put (i);
  Ada.Text_IO.Put_Line;
end loop;

loop
  a := a + 1;
  exit when a = 10;
end loop;

case i is
  when 0 => Ada.Text_IO.Put ("zero");
  when 1 => Ada.Text_IO.Put ("one");
  when 2 => Ada.Text_IO.Put ("two");
  -- case statements have to cover all possible cases:
  when others => Ada.Text_IO.Put ("none of the above");
end case;

for aWeekday in Weekday'Range loop          -- loop over an
enumeration
  Put_Line ( Weekday'Image(AWeekday) );        -- output string

```

```

representation of an enumeration
  if AWeekday in Working_Day then                                -- check of a subtype
of an enumeration
  Put_Line ( " to work for " &
             Working_Hours'Image (Work_Load(aWeekday)) ); -- access
into a lookup table
  end if;
end loop;

```

Packages, procedures and functions

Ada programs consist of packages, procedures and functions.

Example: Package specification (example.ads)

```

package Example is
  type Number is range 1 .. 11;
  procedure Print_and_Increment (j: in out Number);
end Example;

```

Package implementation (example.adb)

```

with Ada.Text_IO;
package body Example is

  i : Number := Number'First;

  procedure Print_and_Increment (j: in out Number) is

    function Next (k: in Number) return Number is
    begin
      return k + 1;
    end Next;

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

-- package initialization executed when the package is made visible
(use clause)
begin
  while i < Number'Last loop
    Print_and_Increment (i);
  end loop;
end Example;

```

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 tasks 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.

```
with Ada.Text_IO; use Ada.Text_IO;

procedure Traffic is

    type Airplane_ID is range 1..10;           -- 10 airplanes

    task type Airplane (ID: Airplane_ID);      -- task representing
airplanes
```

```

type Airplane_Access is access Airplane; -- reference to Airplane

protected type Runway is -- the shared runway
  entry Assign_Aircraft (ID: Airplane_ID);
  entry Cleared_Runway (ID: Airplane_ID);
  entry Wait_For_Clear;
private
  Clear: Boolean := True; -- protected private data - generally
more than just a flag...
end Runway;
type Runway_Access is access all Runway;

-- the air traffic controller takes requests for takeoff and landing
task type Controller (My_Runway: Runway_Access) is
  entry Request_Takeoff (ID: in Airplane_ID; Takeoff: out
Runway_Access);
  entry Request_Approach (ID: in Airplane_ID; Approach: out
Runway_Access);
end Controller;

Runway1      : aliased Runway; -- instantiate a runway
Controller1: Controller (Runway1'Access); -- and a controller to
manage it

----- the implementations of the above types -----
protected body Runway is
  entry Assign_Aircraft (ID: Airplane_ID)
    when Clear is -- the entry guard - tasks are blocked until this
is true
    begin
      Clear := False;
      Put_Line (Airplane_ID'Image (ID) & " on runway ");
    end;

  entry Cleared_Runway (ID: Airplane_ID)
  when not Clear is
    begin
      Clear := True;
      Put_Line (Airplane_ID'Image (ID) & " cleared runway ");
    end;

  entry Wait_For_Clear
  when Clear is
    begin
      null;
    end;
end Runway;

```

```

task body Controller is
begin
  loop
    My_Runway.Wait_For_Clear;      -- wait until runway is available
    select                      -- wait for two types of requests
      when Request_Approach'count = 0 => -- landings have
priority
      accept Request_Takeoff (ID: in Airplane_ID; Takeoff: out
Runway_Access) do
        My_Runway.Assign_Aircraft (ID);   -- reserve runway
        Takeoff := My_Runway;             -- tell airplane which
runway
      end Request_Takeoff;           -- end of the
synchronised part
      or
      accept Request_Approach (ID: in Airplane_ID; Approach: out
Runway_Access) do
        My_Runway.Assign_Aircraft (ID);
        Approach := My_Runway;
      end Request_Approach;
      or                           -- terminate if nobody left who
could call
      terminate;
    end select;
  end loop;
end;

task body Airplane is
  Rwy : Runway_Access;
begin
  Controller1.Request_Takeoff (ID, Rwy); -- wait to be cleared for
takeoff
  Put_Line (Airplane_ID'Image (ID) & " taking off...");
  delay 2.0;
  Rwy.Cleared_Runway (ID);
  delay 5.0; -- fly around a bit...
  loop
    select -- try to request a runway
      Controller1.Request_Approach (ID, Rwy); -- this is a
blocking call
      exit; -- if call returned we're clear for landing -
proceed...
    or
      delay 3.0; -- timeout - if no answer in 3 seconds, do
something else

```

```

        Put_Line (Airplane_ID'Image (ID) & " in holding
pattern");
      end select;
    end loop;
    delay 4.0; -- 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
  for I in Airplane_ID'Range loop -- create a few airplane tasks
    New_Airplane := new Airplane (I);
    delay 3.0;
  end loop;
end Traffic;

```

Pragmas

A pragma is a compiler directive that conveys information to the compiler to allow specific manipulation of compiled output.^[20] Certain pragmas are built in to the language^[21] while other 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 in lieu of a function call (as C/C++ does with inline functions).

References

- [1] Ada 2012 Language Reference Manual (<http://www.adacore.org/standards/ada12.html>) (Draft)
- [2] ISO-8652:1995 (http://www.adaic.org/resources/add_content/standards/95lrm/ARM_HTML/RM-TTL.html)
- [3] ISO/IEC 8652:1995/Amd 1:2007 (<http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=45001>)
- [4] J. Fuegi and J. Francis, "Lovelace & Babbage and the creation of the 1843 'notes'." *Annals of the History of Computing* 25 #4 (October–December 2003): 16-26. Digital Object Identifier (<http://dx.doi.org/10.1109/MAHC.2003.1253887>)
- [5] S. Tucker Taft; Florence Olsen (1999-06-30). "Ada helps churn out less-buggy code" (<http://gen.com/Articles/1999/06/30/Ada-helps-churn-out-lessbuggy-code.aspx>). Government Computer News. pp. 2–3.. Retrieved 2010-09-14.
- [6] Feldman, Michael. "Who's using Ada?" (http://www.seas.gwu.edu/~mfeldman/ada-project-summary.html#Banking_and_Financial_Systems_). SIGAda Education Working Group. .
- [7] <http://www.sei.cmu.edu/about/people/jbg.cfm>
- [8] C.A.R. Hoare, " The Emperor's Old Clothes (<http://zoo.cs.yale.edu/classes/cs422/2011/bib/hoare81emperor.pdf>). " Communications of the ACM, 1981.
- [9] D.A. Watt, B.A. Wichmann and W. Findlay, "Ada: Language and Methodology." Prentice-Hall, 1987.
- [10] J-P. Rosen, "The Ada Paradox(es)", *Ada Letters*, ACM SIGAda, Vol. 24, No. 2, August 2009, pp. 28-35.
- [11] SofTech Inc., Waltham, MA (1983-04-11). "Ada Compiler Validation Summary Report: NYU Ada/ED, Version 19.7 V-001" (<http://oai.dtic.mil/oai/oai?verb=getRecord&metadataPrefix=html&identifier=ADA136759>). . Retrieved 2010-12-16.
- [12] Dewar, Robert B. K.; Fisher Jr., Gerald A.; Schonberg, Edmond; Froelich, Robert; Bryant, Stephen; Goss, Clinton F.; Burke, Michael (November 1980). "The NYU Ada Translator and Interpreter". *ACM SIGPLAN Notices - Proceedings of the ACM-SIGPLAN Symposium on the Ada Programming Language* 15 (11): 194–201. doi:10.1145/948632.948659. ISBN 0-89791-030-3.
- [13] AdaCore. "GNAT Pro Chosen for UK's Next Generation ATC System" (<http://www.adacore.com/2007/06/19/adacore-gnat-pro-chosen-for-uk-next-generation/>). .
- [14] AdaCore. "Look Who's Using Ada" (http://www.adacore.com/home/ada_answers/lookwho/). .
- [15] <http://archive.adaic.com/standards/83lrm/html/Welcome.html>
- [16] <http://www.adaic.org/standards/95lrm/html/RM-TTL.html>
- [17] <http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=45001>

- [18] <http://www.ada-auth.org/standards/12rm/html/RM-TTL.html>
- [19] "Ada Syntax Card" (http://www.digilife.be/quickreferences/QRC/Ada_Syntax_Card.pdf). . Retrieved 28 February 2011.
- [20] United States Government. (<http://archive.adaic.com/standards/83lrm/html/lrm-02-08.html#2.8>) Retrieved April 8, 2011.
- [21] United States Government. (<http://archive.adaic.com/standards/83lrm/html/lrm-B.html>) Retrieved April 8, 2011.

International standards

- ISO/IEC 8652: Information technology—Programming languages—Ada
- ISO/IEC 15291: Information technology—Programming languages—Ada Semantic Interface Specification (ASIS)
- ISO/IEC 18009: Information technology—Programming languages—Ada: Conformity assessment of a language processor (ACATS)
- IEEE Standard 1003.5b-1996, the POSIX Ada binding
- Ada Language Mapping Specification (http://www.omg.org/technology/documents/formal/ada_language_mapping.htm), the CORBA IDL to Ada mapping

Rationale

(These documents have been published in various forms including print.)

- Jean D. Ichbiah, John G. P. Barnes, Robert J. Firth and Mike Woodger, *Rationale for the Design of the Ada Programming Language*, 1986. (<http://archive.adaic.com/standards/83rat/html/Welcome.html>)
- John G. P. Barnes, *Ada 95 rationale : the language : the standard libraries*, 1995. (http://www.adaic.org/resources/add_content/standards/95rat/rat95html/rat95-contents.html)
- John Barnes, *Rationale for Ada 2005*, 2005, 2006. (<http://www.adaic.org/standards/05rat/html/Rat-TTL.html>)

Books

- Grady Booch: *Software Engineering with Ada*, California: The Benjamin/Cummings Publishing Company, Inc., 1987. ISBN 0-8053-0604-8
- Jan Skansholm: *Ada 95 From the Beginning*, Addison-Wesley, ISBN 0-201-40376-5
- Geoff Gilpin: *Ada: A Guided Tour and Tutorial*, Prentice hall, ISBN 978-0-13-004045-9
- John Barnes: *Programming in Ada 2005*, Addison-Wesley, ISBN 0-321-34078-7
- John Barnes: *Programming in Ada plus Language Reference Manual*, Addison-Wesley, ISBN 0-201-56539-0
- John Barnes: *Programming in Ada 95*, Addison-Wesley, ISBN 0-201-34293-6
- John Barnes: *High Integrity Ada: The SPARK Approach*, Addison-Wesley, ISBN 0-201-17517-7
- John Barnes: *High Integrity Software: The SPARK Approach to Safety and Security*, Addison-Wesley, ISBN 0-321-13616-0
- John Beidler: *Data Structures and Algorithms: An Object-Oriented Approach Using Ada 95*, Springer-Verlag, ISBN 0-387-94834-1
- Dean W. Gonzalez: *Ada Programmer's Handbook*, Benjamin-Cummings Publishing Company, ISBN 0-8053-2529-8
- M. Ben-Ari: *Ada for Software Engineers*, John Wiley & Sons, ISBN 0-471-97912-0
- Norman Cohen: *Ada as a Second Language*, McGraw-Hill Science/Engineering/Math, ISBN 0-07-011607-5
- Alan Burns, Andy Wellings: *Real-Time Systems and Programming Languages. Ada 95, Real-Time Java and Real-Time POSIX.*, Addison-Wesley, ISBN 0-201-72988-1
- Alan Burns, Andy Wellings: *Concurrency in Ada*, Cambridge University Press, ISBN 0-521-62911-X
- Colin Atkinson: *Object-Oriented Reuse, Concurrency and Distribution: An Ada-Based Approach*, Addison-Wesley, ISBN 0-201-56527-7
- Grady Booch, Doug Bryan: *Software Engineering with Ada*, Addison-Wesley, ISBN 0-8053-0608-0

- Daniel Stubbs, Neil W. Webre: *Data Structures with Abstract Data Types and Ada*, Brooks Cole, ISBN 0-534-14448-9
- Pascal Ledru: *Distributed Programming in Ada with Protected Objects*, Dissertation.com, ISBN 1-58112-034-6
- Fintan Culwin: *Ada, a Developmental Approach*, Prentice Hall, ISBN 0-13-264680-3
- John English, Fintan Culwin: *Ada 95 the Craft of Object Oriented Programming*, Prentice Hall, ISBN 0-13-230350-7
- David A. Wheeler: *Ada 95*, Springer-Verlag, ISBN 0-387-94801-5
- David R. Musser, Alexander Stepanov: *The Ada Generic Library: Linear List Processing Packages*, Springer-Verlag, ISBN 0-387-97133-5
- Michael B. Feldman: *Software Construction and Data Structures with Ada 95*, Addison-Wesley, ISBN 0-201-88795-9
- Simon Johnston: *Ada 95 for C and C++ Programmers*, Addison-Wesley, ISBN 0-201-40363-3
- Michael B. Feldman, Elliot B. Koffman: *Ada 95*, Addison-Wesley, ISBN 0-201-36123-X
- Nell Dale, Chip Weems, John McCormick: *Programming and Problem Solving with Ada 95*, Jones & Bartlett Publishers, ISBN 0-7637-0293-5
- Nell Dale, John McCormick: *Ada Plus Data Structures: An Object-Oriented Approach, 2nd edition*, Jones & Bartlett Publishers, ISBN 0-7637-3794-1
- Bruce C. Krell: *Developing With Ada: Life-Cycle Methods*, Bantam Dell Pub Group, ISBN 0-553-09102-6
- Judy Bishop: *Distributed Ada: Developments and Experiences*, Cambridge University Press, ISBN 0-521-39251-9
- Bo Sanden: *Software Systems Construction With Examples in Ada*, Prentice Hall, ISBN 0-13-030834-X
- Bruce Hillam: *Introduction to Abstract Data Types Using Ada*, Prentice Hall, ISBN 0-13-045949-6
- David Rudd: *Introduction to Software Design and Development With Ada*, Brooks Cole, ISBN 0-314-02829-3
- Ian C. Pyle: *Developing Safety Systems: A Guide Using Ada*, Prentice Hall, ISBN 0-13-204298-3
- Louis Baker: *Artificial Intelligence With Ada*, McGraw-Hill, ISBN 0-07-003350-1
- Alan Burns, Andy Wellings: *HRT-HOOD: A Structured Design Method for Hard Real-Time Ada Systems*, North-Holland, ISBN 0-444-82164-3
- Walter Savitch, Charles Peterson: *Ada: An Introduction to the Art and Science of Programming*, Benjamin-Cummings Publishing Company, ISBN 0-8053-7070-6
- Mark Allen Weiss: *Data Structures and Algorithm Analysis in Ada*, Benjamin-Cummings Publishing Company, ISBN 0-8053-9055-3
- Henry Ledgard: *ADA: AN INTRODUCTION (Second Edition)*, Springer-Verlag, ISBN 0-387-90814-5
- Dines Bjørner; Ole N. Oest (eds.): *Towards a Formal Description of Ada*, London: Springer-Verlag, 1980. ISBN 3-540-10283-3

Archives

- Ada Programming Language Materials, 1981–1990 (<http://special.lib.umn.edu/findaid/xml/cbi00157.xml>). Charles Babbage Institute, University of Minnesota, Minneapolis.

External links

- Ada programming language/ Ada (programming language) (<http://www.dmoz.org/Computers/Programming/Languages/Ada/>) at the Open Directory Project
- ACM SIGAda (<http://www.sigada.org/>)
- Ada-Europe Organization (<http://www.ada-europe.org/>)
- ISO Home of Ada Standards (<http://www.open-std.org/jtc1/sc22/wg9/>)

- Interview with S.Tucker Taft, Maintainer of Ada (http://www.techworld.com.au/article/223388/z_programming_languages_ada/)

C++

C++

<i>The C++ Programming Language</i> , written by its architect, is the seminal book on the language.	
Paradigm(s)	Multi-paradigm: ^[1] procedural, functional, object-oriented, generic
Appeared in	1983
Designed by	Bjarne Stroustrup
Developer	<ul style="list-style-type: none"> • Bjarne Stroustrup • Bell Labs • ISO/IEC JTC1/SC22/WG21
Stable release	ISO/IEC 14882:2011 (2011)
Typing discipline	Static, unsafe, nominative
Major implementations	C++ Builder, clang, Comeau C/C++, GCC, Intel C++ Compiler, Microsoft Visual C++, Sun Studio
Dialects	Embedded C++, Managed C++, C++/CLI, C++/CX
Influenced by	C, Simula, Ada 83, ALGOL 68, CLU, ML ^[1]
Influenced	Perl, LPC, Lua, Pike, Ada 95, Java, PHP, D, C99, C#, ^[2] Falcon, Seed7
Implementation language	C
OS	Cross-platform (multi-platform)
Usual filename extensions	.h .hh .hpp .hxx .h++ .cc .cpp .cxx .c++

 C++ Programming at Wikibooks

C++ (pronounced "see plus plus") is a statically typed, free-form, multi-paradigm, compiled, general-purpose programming language. It is regarded as an intermediate-level language, as it comprises a combination of both high-level and low-level language features.^[3] Developed by Bjarne Stroustrup starting in 1979 at Bell Labs, it adds object oriented features, such as classes, and other enhancements to the C programming language. Originally named **C with Classes**, the language was renamed C++ in 1983,^[4] as a pun involving the increment operator.

C++ is one of the most popular programming languages^{[5][6]} and is implemented on a wide variety of hardware and operating system platforms. As an efficient compiler to native code, its application domains include systems software, application software, device drivers, embedded software, high-performance server and client applications, and entertainment software such as video games.^[7] Several groups provide both free and proprietary C++ compiler software, including the GNU Project, Microsoft, Intel and Embarcadero Technologies. C++ has greatly influenced many other popular programming languages, most notably C#^[2] and Java. Other successful languages such as Objective-C use a very different syntax and approach to adding classes to C.

C++ is also used for hardware design, where the design is initially described in C++, then analyzed, architecturally constrained, and scheduled to create a register-transfer level hardware description language via high-level synthesis.^[8]

The language began as enhancements to C, first adding classes, then virtual functions, operator overloading, multiple inheritance, templates, and exception handling among other features. After years of development, the C++

programming language standard was ratified in 1998 as *ISO/IEC 14882:1998*. The standard was amended by the 2003 technical corrigendum, *ISO/IEC 14882:2003*. The current standard extending C++ with new features was ratified and published by ISO in September 2011 as *ISO/IEC 14882:2011* (informally known as C++11).^[9]

History



Bjarne Stroustrup, creator of C++

Bjarne Stroustrup began his work on "C with Classes" in 1979.^[4] The idea of creating a new language originated from Stroustrup's experience in programming for his Ph.D. thesis. Stroustrup found that Simula had features that were very helpful for large software development, but the language was too slow for practical use, while BCPL was fast but too low-level to be suitable for large software development. When Stroustrup started working in AT&T Bell Labs, he had the problem of analyzing the UNIX kernel with respect to distributed computing. Remembering his Ph.D. experience, Stroustrup set out to enhance the C language with Simula-like features. C was chosen because it was general-purpose, fast, portable and widely used.

Besides C and Simula, some other languages that inspired him were ALGOL 68, Ada, CLU and ML. At first, the class, derived class, strong type checking, inlining, and default argument features were added to C via Stroustrup's C++ to C compiler, Cfront. The first commercial implementation of C++ was released on 14 October 1985.^[4]

In 1983, the name of the language was changed from *C with Classes* to C++ (++ being the increment operator in C). New features were added including virtual functions, function name and operator overloading, references, constants, user-controlled free-store memory control, improved type checking, and BCPL style single-line comments with two forward slashes (//). In 1985, the first edition of *The C++ Programming Language* was released, providing an important reference to the language, since there was not yet an official standard.^[10] Release 2.0 of C++ came in 1989 and the updated second edition of *The C++ Programming Language* was released in 1991.^[11] New features included multiple inheritance, abstract classes, static member functions, const member functions, and protected members. In 1990, *The Annotated C++ Reference Manual* was published. This work became the basis for the future standard. Late feature additions included templates, exceptions, namespaces, new casts, and a Boolean type.

As the C++ language evolved, the standard library evolved with it. The first addition to the C++ standard library was the stream I/O library which provided facilities to replace the traditional C functions such as printf and scanf. Later, among the most significant additions to the standard library, was large amounts of the Standard Template Library.

C++ is sometimes called a hybrid language.^[12]

It is possible to write object oriented or procedural code in the same program in C++. This has caused some concern that some C++ programmers are still writing procedural code, but are under the impression that it is object oriented, simply because they are using C++. Often it is an amalgamation of the two. This usually causes most problems when the code is revisited or the task is taken over by another coder.^[13]

C++ continues to be used and is one of the preferred programming languages to develop professional applications.^[14]

Etymology

According to Stroustrup: "the name signifies the evolutionary nature of the changes from C".^[15] During C++'s development period, the language had been referred to as "new C", then "C with Classes". The final name is credited to Rick Mascitti (mid-1983) and was first used in December 1983. When Mascitti was questioned informally in 1992 about the naming, he indicated that it was given in a tongue-in-cheek spirit. It stems from C's "++" operator (which increments the value of a variable) and a common naming convention of using "+" to indicate an enhanced computer

program. There is no language called "C plus". ABCL/c+ was the name of an earlier, unrelated programming language.

Standardization

Year	C++ Standard	Informal name
1998	ISO/IEC 14882:1998 ^[16]	C++98
2003	ISO/IEC 14882:2003 ^[17]	C++03
2007	ISO/IEC TR 19768:2007 ^[18]	C++TR1
2011	ISO/IEC 14882:2011 ^[19]	C++11

In 1998, the C++ standards committee (the ISO/IEC JTC1/SC22/WG21 working group) standardized C++ and published the international standard *ISO/IEC 14882:1998* (informally known as *C++98*). For some years after the official release of the standard, the committee processed defect reports, and in 2003 published a corrected version of the C++ standard, *ISO/IEC 14882:2003*. In 2005, a technical report, called the "Library Technical Report 1" (often known as TR1 for short), was released. While not an official part of the standard, it specified a number of extensions to the standard library, which were expected to be included in the next version of C++.

The latest major revision of the C++ standard, C++11, (formerly known as C++0x) was approved by ISO/IEC on 12 August 2011.^[20] It has been published as 14882:2011.^[21]

Philosophy

In *The Design and Evolution of C++* (1994), Bjarne Stroustrup describes some rules that he used for the design of C++:

- C++ is designed to be a statically typed, general-purpose language that is as efficient and portable as C
- C++ is designed to directly and comprehensively support multiple programming styles (procedural programming, data abstraction, object-oriented programming, and generic programming)
- C++ is designed to give the programmer choice, even if this makes it possible for the programmer to choose incorrectly
- C++ is designed to be as compatible with C as much as possible, therefore providing a smooth transition from C
- C++ avoids features that are platform specific or not general purpose
- C++ does not incur overhead for features that are not used (the "zero-overhead principle")
- C++ is designed to function without a sophisticated programming environment

Inside the C++ Object Model (Lippman, 1996) describes how compilers may convert C++ program statements into an in-memory layout. Compiler authors are, however, free to implement the standard in their own manner.

Standard library

The 1998 ANSI/ISO C++ standard consists of two parts: the core language and the C++ Standard Library; the latter includes most of the Standard Template Library (STL) and a slightly modified version of the C standard library. Many C++ libraries exist that are not part of the standard, and, using linkage specification, libraries can even be written in languages such as BASIC, C, Fortran, or Pascal. Which of these are supported is compiler-dependent.

The C++ standard library incorporates the C standard library with some small modifications to make it optimized with the C++ language. Another large part of the C++ library is based on the STL. This provides such useful tools as containers (for example vectors and lists), iterators to provide these containers with array-like access and algorithms to perform operations such as searching and sorting. Furthermore (multi)maps (associative arrays) and (multi)sets are provided, all of which export compatible interfaces. Therefore it is possible, using templates, to write generic algorithms that work with any container or on any sequence defined by iterators. As in C, the features of the library are accessed by using the `#include` directive to include a standard header. C++ provides 105 standard headers, of which 27 are deprecated.

The STL was originally a third-party library from HP and later SGI, before its incorporation into the C++ standard. The main architect behind STL is Alexander Stepanov, who experimented with generic algorithms and containers for many years. When he started with C++, he finally found a language where it was possible to create generic algorithms (e.g. STL sort) that perform even better than e.g. the C standard library `qsort`, thanks to C++ features like using inlining and compile-time binding instead of function pointers. The standard does not refer to it as "STL", as it is merely a part of the standard library, but many people still use that term to distinguish it from the rest of the library (input/output streams, internationalization, diagnostics, the C library subset, etc.).

Most C++ compilers provide an implementation of the C++ standard library, including the STL. Compiler-independent implementations of the STL, such as STLPort,^[22] also exist. Other projects also produce various custom implementations of the C++ standard library and the STL with various design goals.

Language features

C++ inherits most of C's syntax. The following is Bjarne Stroustrup's version of the Hello world program that uses the C++ Standard Library stream facility to write a message to standard output:^{[23][24]}

```
# include <iostream>

int main()
{
    std::cout << "Hello, world!\n";
}
```

Within functions that define a non-void return type, failure to return a value before control reaches the end of the function results in undefined behaviour (compilers typically provide the means to issue a diagnostic in such a case).^[25] The sole exception to this rule is the `main` function, which implicitly returns a value of zero.^[26]

Operators and operator overloading

Operators that cannot be overloaded

Operator	Symbol
Scope resolution operator	<code>::</code>
Conditional operator	<code>? :</code>
dot operator	<code>.</code>
Member selection operator	<code>. *</code>
"sizeof" operator	<code>sizeof</code>
"typeid" operator	<code>typeid</code>

C++ provides more than 35 operators, covering basic arithmetic, bit manipulation, indirection, comparisons, logical operations and others. Almost all operators can be overloaded for user-defined types, with a few notable exceptions such as member access (`.` and `.*`) as well as the conditional operator. The rich set of overloadable operators is central to using C++ as a domain-specific language. The overloadable operators are also an essential part of many advanced C++ programming techniques, such as smart pointers. Overloading an operator does not change the precedence of calculations involving the operator, nor does it change the number of operands that the operator uses (any operand may however be ignored by the operator, though it will be evaluated prior to execution). Overloaded "`&&`" and "`||`" operators lose their short-circuit evaluation property.

Templates

C++ templates enable generic programming. C++ supports both function and class templates. Templates may be parameterized by types, compile-time constants, and other templates. C++ templates are implemented by *instantiation* at compile-time. To instantiate a template, compilers substitute specific arguments for a template's parameters to generate a concrete function or class instance. Some substitutions are not possible; these are eliminated by an overload resolution policy described by the phrase "Substitution failure is not an error" (SFINAE). Templates are a powerful tool that can be used for generic programming, template metaprogramming, and code optimization, but this power implies a cost. Template use may increase code size, since each template instantiation produces a copy of the template code: one for each set of template arguments. This is in contrast to run-time generics seen in other languages (e.g. Java) where at compile-time the type is erased and a single template body is preserved.

Templates are different from macros: while both of these compile-time language features enable conditional compilation, templates are not restricted to lexical substitution. Templates are aware of the semantics and type system of their companion language, as well as all compile-time type definitions, and can perform high-level operations including programmatic flow control based on evaluation of strictly type-checked parameters. Macros are capable of conditional control over compilation based on predetermined criteria, but cannot instantiate new types, recurse, or perform type evaluation and in effect are limited to pre-compilation text-substitution and text-inclusion/exclusion. In other words, macros can control compilation flow based on pre-defined symbols but cannot, unlike templates, independently instantiate new symbols. Templates are a tool for static polymorphism (see below) and generic programming.

In addition, templates are a compile time mechanism in C++ that is Turing-complete, meaning that any computation expressible by a computer program can be computed, in some form, by a template metaprogram prior to runtime.

In summary, a template is a compile-time parameterized function or class written without knowledge of the specific arguments used to instantiate it. After instantiation, the resulting code is equivalent to code written specifically for the passed arguments. In this manner, templates provide a way to decouple generic, broadly applicable aspects of functions and classes (encoded in templates) from specific aspects (encoded in template parameters) without

sacrificing performance due to abstraction.

Objects

C++ introduces object-oriented programming (OOP) features to C. It offers classes, which provide the four features commonly present in OOP (and some non-OOP) languages: abstraction, encapsulation, inheritance, and polymorphism. One distinguishing feature of C++ classes compared to classes in other programming languages is support for deterministic destructors, which in turn provide support for the Resource Acquisition is Initialization concept.

Encapsulation

Encapsulation is the hiding of information in order to ensure that data structures and operators are used as intended and to make the usage model more obvious to the developer. C++ provides the ability to define classes and functions as its primary encapsulation mechanisms. Within a class, members can be declared as either public, protected, or private in order to explicitly enforce encapsulation. A public member of the class is accessible to any function. A private member is accessible only to functions that are members of that class and to functions and classes explicitly granted access permission by the class ("friends"). A protected member is accessible to members of classes that inherit from the class in addition to the class itself and any friends.

The OO principle is that all of the functions (and only the functions) that access the internal representation of a type should be encapsulated within the type definition. C++ supports this (via member functions and friend functions), but does not enforce it: the programmer can declare parts or all of the representation of a type to be public, and is allowed to make public entities that are not part of the representation of the type. Therefore, C++ supports not just OO programming, but other weaker decomposition paradigms, like modular programming.

It is generally considered good practice to make all data private or protected, and to make public only those functions that are part of a minimal interface for users of the class. This can hide the details of data implementation, allowing the designer to later fundamentally change the implementation without changing the interface in any way.^{[27][28]}

Inheritance

Inheritance allows one data type to acquire properties of other data types. Inheritance from a base class may be declared as public, protected, or private. This access specifier determines whether unrelated and derived classes can access the inherited public and protected members of the base class. Only public inheritance corresponds to what is usually meant by "inheritance". The other two forms are much less frequently used. If the access specifier is omitted, a "class" inherits privately, while a "struct" inherits publicly. Base classes may be declared as virtual; this is called virtual inheritance. Virtual inheritance ensures that only one instance of a base class exists in the inheritance graph, avoiding some of the ambiguity problems of multiple inheritance.

Multiple inheritance is a C++ feature not found in most other languages, allowing a class to be derived from more than one base classes; this allows for more elaborate inheritance relationships. For example, a "Flying Cat" class can inherit from both "Cat" and "Flying Mammal". Some other languages, such as C# or Java, accomplish something similar (although more limited) by allowing inheritance of multiple interfaces while restricting the number of base classes to one (interfaces, unlike classes, provide only declarations of member functions, no implementation or member data). An interface as in C# and Java can be defined in C++ as a class containing only pure virtual functions, often known as an abstract base class or "ABC". The member functions of such an abstract base class are normally explicitly defined in the derived class, not inherited implicitly. C++ virtual inheritance exhibits an ambiguity resolution feature called dominance.

Polymorphism

Polymorphism enables one common interface for many implementations, and for objects to act differently under different circumstances.

C++ supports several kinds of *static* (compile-time) and *dynamic* (run-time) polymorphisms. Compile-time polymorphism does not allow for certain run-time decisions, while run-time polymorphism typically incurs a performance penalty.

Static polymorphism

Function overloading allows programs to declare multiple functions having the same name (but with different arguments). The functions are distinguished by the number or types of their formal parameters. Thus, the same function name can refer to different functions depending on the context in which it is used. The type returned by the function is not used to distinguish overloaded functions and would result in a compile-time error message.

When declaring a function, a programmer can specify for one or more parameters a default value. Doing so allows the parameters with defaults to optionally be omitted when the function is called, in which case the default arguments will be used. When a function is called with fewer arguments than there are declared parameters, explicit arguments are matched to parameters in left-to-right order, with any unmatched parameters at the end of the parameter list being assigned their default arguments. In many cases, specifying default arguments in a single function declaration is preferable to providing overloaded function definitions with different numbers of parameters.

Templates in C++ provide a sophisticated mechanism for writing generic, polymorphic code. In particular, through the Curiously Recurring Template Pattern, it's possible to implement a form of static polymorphism that closely mimics the syntax for overriding virtual functions. Since C++ templates are type-aware and Turing-complete, they can also be used to let the compiler resolve recursive conditionals and generate substantial programs through template metaprogramming. Contrary to some opinion, template code will not generate a bulk code after compilation with the proper compiler settings.^[29]

Dynamic polymorphism

Inheritance

Variable pointers (and references) to a base class type in C++ can refer to objects of any derived classes of that type in addition to objects exactly matching the variable type. This allows arrays and other kinds of containers to hold pointers to objects of differing types. Because assignment of values to variables usually occurs at run-time, this is necessarily a run-time phenomenon.

C++ also provides a `dynamic_cast` operator, which allows the program to safely attempt conversion of an object into an object of a more specific object type (as opposed to conversion to a more general type, which is always allowed). This feature relies on run-time type information (RTTI). Objects known to be of a certain specific type can also be cast to that type with `static_cast`, a purely compile-time construct that is faster and does not require RTTI.

Virtual member functions

Ordinarily, when a function in a derived class overrides a function in a base class, the function to call is determined by the type of the object. A given function is overridden when there exists no difference in the number or type of parameters between two or more definitions of that function. Hence, at compile time, it may not be possible to determine the type of the object and therefore the correct function to call, given only a base class pointer; the decision is therefore put off until runtime. This is called dynamic dispatch. Virtual member functions or *methods*^[30] allow the most specific implementation of the function to be called, according to the actual run-time type of the object. In C++ implementations, this is commonly done using virtual function tables. If the object type is known, this may be bypassed by prepending a fully qualified class name before the function call, but in general calls to virtual

functions are resolved at run time.

In addition to standard member functions, operator overloads and destructors can be virtual. A general rule of thumb is that if any functions in the class are virtual, the destructor should be as well. As the type of an object at its creation is known at compile time, constructors, and by extension copy constructors, cannot be virtual. Nonetheless a situation may arise where a copy of an object needs to be created when a pointer to a derived object is passed as a pointer to a base object. In such a case, a common solution is to create a `clone()` (or similar) virtual function that creates and returns a copy of the derived class when called.

A member function can also be made "pure virtual" by appending it with `= 0` after the closing parenthesis and before the semicolon. A class containing a pure virtual function is called an *abstract data type*. Objects cannot be created from abstract data types; they can only be derived from. Any derived class inherits the virtual function as pure and must provide a non-pure definition of it (and all other pure virtual functions) before objects of the derived class can be created. A program that attempts to create an object of a class with a pure virtual member function or inherited pure virtual member function is ill-formed.

Parsing and processing C++ source code

It is relatively difficult to write a good C++ parser with classic parsing algorithms such as LALR(1).^[31] This is partly because the C++ grammar is not LALR. Because of this, there are very few tools for analyzing or performing non-trivial transformations (e.g., refactoring) of existing code. One way to handle this difficulty is to choose a different syntax. More powerful parsers, such as GLR parsers, can be substantially simpler (though slower).

Parsing (in the literal sense of producing a syntax tree) is not the most difficult problem in building a C++ processing tool. Such tools must also have the same understanding of the meaning of the identifiers in the program as a compiler might have. Practical systems for processing C++ must then not only parse the source text, but be able to resolve for each identifier precisely which definition applies (e.g. they must correctly handle C++'s complex scoping rules) and what its type is, as well as the types of larger expressions.

Finally, a practical C++ processing tool must be able to handle the variety of C++ dialects used in practice (such as that supported by the GNU Compiler Collection and that of Microsoft's Visual C++) and implement appropriate analyzers, source code transformers, and regenerate source text. Combining advanced parsing algorithms such as GLR with symbol table construction and program transformation machinery can enable the construction of arbitrary C++ tools.

Compatibility

Producing a reasonably standards-compliant C++ compiler has proven to be a difficult task for compiler vendors in general. For many years, different C++ compilers implemented the C++ language to different levels of compliance to the standard, and their implementations varied widely in some areas such as partial template specialization. Recent releases of most popular C++ compilers support almost all of the C++ 1998 standard.^[32]

In order to give compiler vendors greater freedom, the C++ standards committee decided not to dictate the implementation of name mangling, exception handling, and other implementation-specific features. The downside of this decision is that object code produced by different compilers is expected to be incompatible. There were, however, attempts to standardize compilers for particular machines or operating systems (for example C++ ABI),^[33] though they seem to be largely abandoned now.

Exported templates

One particular point of contention is the `export` keyword, intended to allow template definitions to be separated from their declarations. The first widely available compiler to implement `export` was Comeau C/C++, in early 2003 (five years after the release of the standard); in 2004, the beta compiler of Borland C++ Builder X was also released with `export`. Both of these compilers are based on the EDG C++ front end. Other compilers such as GCC do not support it at all. *Beginning ANSI C++* by Ivor Horton provides example code with the keyword that will not compile in most compilers, without reference to this problem. Herb Sutter, former convener of the C++ standards committee, recommended that `export` be removed from future versions of the C++ standard.^[34] During the March 2010 ISO C++ standards meeting, the C++ standards committee voted to remove exported templates entirely from C++11, but reserve the keyword for future use.^[35]

With C

C++ is often considered to be a superset of C, but this is not strictly true.^[36] Most C code can easily be made to compile correctly in C++, but there are a few differences that cause some valid C code to be invalid or behave differently in C++.

One commonly encountered difference is that C allows implicit conversion from `void*` to other pointer types, but C++ does not. Another common portability issue is that C++ defines many new keywords, such as `new` and `class`, that may be used as identifiers (e.g. variable names) in a C program.

Some incompatibilities have been removed by the 1999 revision of the C standard (C99), which now supports C++ features such as line comments (`//`), and mixed declarations and code. On the other hand, C99 introduced a number of new features that C++ did not support, such as variable-length arrays, native complex-number types, designated initializers, and compound literals.^[37] However, at least some of the C99-introduced features were included in the subsequent version of the C++ standard, C++11:^{[38][39]}

- C99 preprocessor (including variadic macros, wide/narrow literal concatenation, wider integer arithmetic)
- `_Pragma()`
- `long long`
- `__func__`
- Headers:
 - `cstdbool` (`stdbool.h`)
 - `cstdint` (`stdint.h`)
 - `cinttypes` (`inttypes.h`).

In order to intermix C and C++ code, any function declaration or definition that is to be called from/used both in C and C++ must be declared with C linkage by placing it within an `extern "C" /*...*/` block. Such a function may not rely on features depending on name mangling (i.e., function overloading).

Criticism

Due to its large feature set and oft-perceived "strict" syntax, the language is sometimes criticized as being overly complicated and thus difficult to fully master.^[40]

C++ is sometimes compared unfavorably with more strictly object-oriented languages on the basis that it enables programmers to "mix and match" declarative, functional, generic, modular, and procedural programming styles with object-oriented programming, rather than strictly enforcing a single style, although C++ is intentionally a multi-paradigm language.^[1]

A widely distributed satirical article portrayed Bjarne Stroustrup, interviewed for a 1998 issue of IEEE's *Computer* magazine,^[41] confessing that C++ was deliberately designed to be complex and difficult, weeding out amateur programmers and raising the salaries of the few programmers who could master the language. The FAQ section of

Stroustrup's personal website contains a denial and a link to the actual interview.^[42]

Richard Stallman criticizes C++ for having ambiguous grammar and "gratuitous, trivial incompatibilities with C [...] that are of no *great* benefit".^[43]

Linus Torvalds has said, "C++ is a horrible language. It's made more horrible by the fact that a lot of substandard programmers use it".^[44]

Finally, several authors have remarked that C++ is not a true object-oriented language.

The same problem occurs with programming languages. As stated earlier, many C programmers moved into the realm of object orientation by migrating to C++ before being directly exposed to OO concepts. This would always come out in an interview. Many times developers who claim to be C++ programmers are simply C programmers using C++ compilers. [...] We have already mentioned that C++ is not a true object-oriented programming language but is actually an object-based programming language. Remember that C++ is considered to be object-based. Object-oriented concepts are not enforced. You can write a non-object-oriented C program using a C++ compiler.

—Matt Weisfeld,^[45]

It is interesting to see what is being done out in the world under the name object-oriented. I have been shown some very, very strange looking pieces of code over the years, by various people, including people in universities, that they have said is OOP code written in an OOP language—and actually I made up the term object-oriented, and I can tell you, I didn't have C++ in mind.

—Alan Kay, Smalltalk co-creator,^[46]

References

- [1] Stroustrup, Bjarne (1997). "1". *The C++ Programming Language* (Third ed.). ISBN 0-201-88954-4. OCLC 59193992.
- [2] Naugler, David (May 2007). "C# 2.0 for C++ and Java programmer: conference workshop". *Journal of Computing Sciences in Colleges* 22 (5). "Although C# has been strongly influenced by Java it has also been strongly influenced by C++ and is best viewed as a descendant of both C++ and Java."
- [3] Schildt, Herbert (1 August 1998). *C++ The Complete Reference* (Third ed.). Osborne McGraw-Hill. ISBN 978-0-07-882476-0.
- [4] Stroustrup, Bjarne (7 March 2010). "C++ FAQ: When was C++ Invented" (http://www2.research.att.com/~bs/bs_faq.html#invention). ATT.com. . Retrieved 16 September 2010.
- [5] "Programming Language Popularity" (<http://www.langpop.com/>). 2009. . Retrieved 16 January 2009.
- [6] "TIOBE Programming Community Index" (<http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>). 2009. . Retrieved 3 August 2011.
- [7] C++ Applications (<http://www2.research.att.com/~bs/applications.html>)
- [8] "What's CvSDL?" (<http://www.cvsdl.com/>). : c_sdl. . Retrieved 8 March 2010. "CvSDL was introduced in 2003 as a C++ class framework with Verilog features that worked like a Verilog simulator. Since then it has been revamped to be a standard-compliant HDL simulator, currently supporting Verilog. It is capable of cosimulating with SystemC. It can be used just as an HDL simulator or to generate executable specifications written in Verilog and SystemC on the hardware side and in C, C++ and SystemC on the software side."
- [9] "ISO/IEC 14882:2011" (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=50372). ISO. . Retrieved 3 September 2011.
- [10] Stroustrup, Bjarne. "The C++ Programming Language" (<http://www2.research.att.com/~bs/1st.html>). . Retrieved 16 September 2010.
- [11] Stroustrup, Bjarne. "The C++ Programming Language" (<http://www2.research.att.com/~bs/2nd.html>). . Retrieved 16 September 2010.
- [12] Voegele, Jason. "Programming Language Comparison" (<http://www.jvoegele.com/software/langcomp.html>). . Retrieved 7 April 2011.
- [13] Bhatti, M. U.; Ducasse, S.; Rashid, A. (June 2008). "Aspect Mining in Procedural Object Oriented Code" (<http://scg.unibe.ch/archive/external/Bhat08a-ICPC2008-AspectMining.pdf>). *Proceedings of the International Conference on Program Comprehension*: 230–235. doi:10.1109/ICPC.2008.45. . Retrieved 19 December 2011.
- [14] "Most Popular Programming Languages" (<http://langpop.com>). . Retrieved 7 September 2011.
- [15] "Bjarne Stroustrup's FAQ – Where did the name "C++" come from?" (http://public.research.att.com/~bs/bs_faq.html#name). . Retrieved 16 January 2008.
- [16] "ISO/IEC 14882:1998" (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=25845). .
- [17] "ISO/IEC 14882:2003" (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=38110). .

- [18] "ISO/IEC TR 19768:2007" (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=43289)..
- [19] "ISO/IEC 14882:2011" (http://www.iso.org/iso/iso_catalogue/catalogue_ics/catalogue_detail_ics.htm?ics1=35&ics2=60&ics3=&csnumber=50372)..
- [20] <http://herbsutter.com/2011/08/12/we-have-an-international-standard-c0x-is-unanimously-approved/>
- [21] "ISO/IEC 14882:2011" (http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=50372). ISO. 2 September 2011.. Retrieved 3 September 2011.
- [22] STLPort home page (<http://www.stlport.org/>), quote from "The C++ Standard Library" by Nicolai M. Josuttis, p138., ISBN 0-201-37926-0, Addison-Wesley, 1999: "An exemplary version of STL is the STLport, which is available for free for any platform"
- [23] Stroustrup, Bjarne (2000). *The C++ Programming Language* (Special ed.). Addison-Wesley. p. 46. ISBN 0-201-70073-5.
- [24] Open issues for The C++ Programming Language (3rd Edition) (http://www.research.att.com/~bs/3rd_issues.html) – This code is copied directly from Bjarne Stroustrup's errata page (p. 633). He addresses the use of '\n' rather than std::endl. Also see [www.research.att.com \(~bs/bs_faq2.html#void-main\)](http://www.research.att.com/~bs/bs_faq2.html#void-main) for an explanation of the implicit return 0; in the main function. This implicit return is not available in other functions.
- [25] ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages – C++ §6.6.3 The return statement [stmt.return]* para. 2
- [26] ISO/IEC (2003). *ISO/IEC 14882:2003(E): Programming Languages – C++ §3.6.1 Main function [basic.start.main]* para. 5
- [27] Sutter, Herb; Alexandrescu, Andrei (2004). *C++ Coding Standards: 101 Rules, Guidelines, and Best Practices*. Addison-Wesley.
- [28] Henricson, Mats; Nyquist, Erik (1997). *Industrial Strength C++*. Prentice Hall. ISBN 0-13-120965-5.
- [29] "Nobody Understands C++: Part 5: Template Code Bloat" (<http://blog.emptycrate.com/node/307>). <http://blog.emptycrate.com/>: EmptyCrate Software. Travel. Stuff.. 6 May 2008.. Retrieved 8 March 2010. "On occasion you will read or hear someone talking about C++ templates causing code bloat. I was thinking about it the other day and thought to myself, "self, if the code does exactly the same thing then the compiled code cannot really be any bigger, can it?" [...] And what about compiled code size? Each were compiled with the command g++ <filename>.cpp -O3. Non-template version: 8140 bytes, template version: 8028 bytes!"
- [30] Stroustrup, Bjarne (2000). *The C++ Programming Language* (Special ed.). Addison-Wesley. p. 310. ISBN 0-201-70073-5. "A virtual member function is sometimes called a method."
- [31] Birkett, Andrew. "Parsing C++ at nobugs.org" (<http://www.nobugs.org/developer/parsingcpp/>). Nobugs.org.. Retrieved 3 July 2009.
- [32] Sutter, Herb (15 April 2003). "C++ Conformance Roundup" (<http://www.ddj.com/dept/cpp/184401381>). *Dr. Dobb's Journal*.. Retrieved 30 May 2006.
- [33] "C++ ABI" (<http://www.codesourcery.com/cxx-abi/>).. Retrieved 30 May 2006.
- [34] Why We Can't Afford Export (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf>) PDF (266 KB)
- [35] Sutter, Herb (13 March 2010). "Trip Report: March 2010 ISO C++ Standards Meeting" (<http://herbsutter.com/2010/03/13/trip-report-march-2010-iso-c-standards-meeting/>).. Retrieved 8 April 2010.
- [36] "Bjarne Stroustrup's FAQ – Is C a subset of C++?" (http://public.research.att.com/~bs/bs_faq.html#C-is-subset).. Retrieved 18 January 2008.
- [37] "C9X – The New C Standard" (http://home.datacomm.ch/t_wolf/tw/c/c9x_changes.html).. Retrieved 27 December 2008.
- [38] "C++0x Support in GCC" (<http://gcc.gnu.org/projects/cxx0x.html>).. Retrieved 12 October 2010.
- [39] "C++0x Core Language Features In VC10: The Table" (<http://blogs.msdn.com/b/vcblog/archive/2010/04/06/c-0x-core-language-features-in-vc10-the-table.aspx>).. Retrieved 12 October 2010.
- [40] Morris, Richard (2 July 2009). "Niklaus Wirth: Geek of the Week" (<http://www.simple-talk.com/opinion/geek-of-the-week/niklaus-wirth-geek-of-the-week/>).. Retrieved 8 August 2009. "C++ is a language that was designed to cater to everybody's perceived needs. As a result, the language and even more so its implementations have become complex and bulky, difficult to understand, and likely to contain errors for ever."
- [41] Unattributed. Previously unpublished interview with Bjarne Stroustrup, designer of C++ (http://harmful.cat-v.org/software/c++/I_did_it_for_you_all).
- [42] Stroustrup, Bjarne. Stroustrup FAQ: Did you really give an interview to IEEE? (http://www2.research.att.com/~bs/bs_faq.html#IEEE)
- [43] "Stallman Lecture in Lund, Sweden 2000-02-11, part 2, transcription" (<http://ftp.fukt.bsnet.se/pub/movies/stallman/stallman2c.sub>). Datorföreningen vid Lunds universitet och Lunds tekniska högskola. 11 February 2000.. Retrieved 16 January 2010.
- [44] Torvalds, Linus (6 September 2007). "Re: [RFC] Convert builtin-mailinfo.c to use The Better String Library" (<http://thread.gmane.org/gmane.comp.version-control.git/57643/focus=57918>). git mailing list.. Retrieved 15 March 2011.
- [45] Weisfeld, Matt (2009). *The Object-Oriented Thought Process* (<http://books.google.ca/books?id=9PIoRAbsmBMC&pg=PT350>) (third ed.). Pearson Education. .
- [46] Kay, Alan (1997). *Alan Kay: The Computer Revolution Hasn't Happened Yet. Keynote OOPSLA 1997* (<http://video.google.com/videoplay?docid=-2950949730059754521>). Event occurs at 10:00. .

Further reading

- Abrahams, David; Gurtovoy, Aleksey. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley. ISBN 0-321-22725-5.
- Alexandrescu, Andrei (2001). *Modern C++ Design: Generic Programming and Design Patterns Applied*. Addison-Wesley. ISBN 0-201-70431-5.
- Alexandrescu, Andrei; Sutter, Herb (2004). *C++ Design and Coding Standards: Rules and Guidelines for Writing Programs*. Addison-Wesley. ISBN 0-321-11358-6.
- Becker, Pete (2006). *The C++ Standard Library Extensions : A Tutorial and Reference*. Addison-Wesley. ISBN 0-321-41299-0.
- Brokken, Frank (2010). *C++ Annotations* (<http://www.icce.rug.nl/documents/cplusplus/>). University of Groningen. ISBN 90-367-0470-7.
- Coplien, James O. (1992, reprinted with corrections 1994). *Advanced C++: Programming Styles and Idioms*. ISBN 0-201-54855-0.
- Dewhurst, Stephen C. (2005). *C++ Common Knowledge: Essential Intermediate Programming*. Addison-Wesley. ISBN 0-321-32192-8.
- Information Technology Industry Council (15 October 2003). *Programming languages – C++* (Second ed.). Geneva: ISO/IEC. 14882:2003(E).
- Josuttis, Nicolai M. (2012). *The C++ Standard Library, A Tutorial and Reference* (Second ed.). Addison-Wesley. ISBN 0-321-62321-5.
- Koenig, Andrew; Moo, Barbara E. (2000). *Accelerated C++ – Practical Programming by Example*. Addison-Wesley. ISBN 0-201-70353-X.
- Lippman, Stanley B.; Lajoie, Josée; Moo, Barbara E. (2011). *C++ Primer* (Fifth ed.). Addison-Wesley. ISBN 0-470-93244-9.
- Lippman, Stanley B. (1996). *Inside the C++ Object Model*. Addison-Wesley. ISBN 0-201-83454-5.
- Meyers, Scott (2005). *Effective C++* (Third ed.). Addison-Wesley. ISBN 0-321-33487-6.
- Stroustrup, Bjarne (2000). *The C++ Programming Language* (Special ed.). Addison-Wesley. ISBN 0-201-70073-5.
- Stroustrup, Bjarne (1994). *The Design and Evolution of C++*. Addison-Wesley. ISBN 0-201-54330-3.
- Stroustrup, Bjarne (2009). *Programming Principles and Practice Using C++*. Addison-Wesley. ISBN 0-321-54372-6.
- Sutter, Herb (2001). *More Exceptional C++: 40 New Engineering Puzzles, Programming Problems, and Solutions*. Addison-Wesley. ISBN 0-201-70434-X.
- Sutter, Herb (2004). *Exceptional C++ Style*. Addison-Wesley. ISBN 0-201-76042-8.
- Vandevoorde, David; Josuttis, Nicolai M. (2003). *C++ Templates: The complete Guide*. Addison-Wesley. ISBN 0-201-73484-2.

External links

- JTC1/SC22/WG21 (<http://www.open-std.org/jtc1/sc22/wg21/>) – The ISO/IEC C++ Standard Working Group
 - n3242.pdf (<http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf>) – Final Committee Draft of "ISO/IEC IS 14882 – Programming Languages – C++" (28 February 2011)
- A paper by Stroustrup showing the timeline of C++ evolution (1979–1991) (<http://www.research.att.com/~bs/hopl2.pdf>)
- Bjarne Stroustrup's C++ Style and Technique FAQ (http://www.research.att.com/~bs/bs_faq2.html)
- C++ FAQ Lite by Marshall Cline (<http://www.parashift.com/c++-faq-lite/>)
- C++ FQA Lite - Yossi Kreinin (<http://yosefk.com/c++fqa/>)
- Hamilton, Naomi (25 June 2008). "The A-Z of Programming Languages: C++" (http://www.computerworld.com.au/article/250514/-z_programming_languages_c?pp=1&fp=16&fpid=1). *Computerworld*. Interview with Bjarne Stroustrup.
- Kalev, Danny (15 August 2008). "The State of the Language: An Interview with Bjarne Stroustrup" (<http://www.devx.com/SpecialReports/Article/38813/0/page/1>). *DevX* (QuinStreet Inc.).
- Katdare, Kaustubh (1 February 2008). "Dr. Bjarne Stroustrup – Inventor of C++" (<http://www.crazyengineers.com/dr-bjarne-stroustrup-inventor-of-c/>). *CrazyEngineers*.
- Code practices for not breaking binary compatibility between releases of C++ libraries (http://techbase.kde.org/Policies/Binary_Compatibility_Issues_With_C++) (from KDE Techbase)

That's a duck

Duck typing

<noinclude Mm q q q Vd XT9 </noinclude> In computer programming with object-oriented programming languages, **duck typing** is a style of dynamic typing in which an object's *methods and properties* determine the valid semantics, rather than its inheritance from a particular class or implementation of a specific interface. The name of the concept refers to the duck test, attributed to James Whitcomb Riley (see history below), which may be phrased as follows:

When I see a bird that walks like a duck and swims like a duck and quacks like a duck, I call that bird a duck.^[1]

In duck typing, one is concerned with just those aspects of an object that are used, rather than with the type of the object itself. For example, in a non-duck-typed language, one can create a function that takes an object of type Duck and calls that object's walk and quack methods. In a duck-typed language, the equivalent function would take an object of any type and call that object's walk and quack methods. If the object does not have the methods that are called then the function signals a run-time error. If the object does have the methods, then they are executed no matter the type of the object, evoking the quotation and hence the name of this form of typing.

Duck typing is aided by habitually *not* testing for the type of arguments in method and function bodies, relying on documentation, clear code and testing to ensure correct use.

Concept examples

Consider the following pseudo-code for a duck-typed language:

```
function calculate(a, b, c) => return (a+b)*c

example1 = calculate (1, 2, 3)
example2 = calculate ([1, 2, 3], [4, 5, 6], 2)
example3 = calculate ('apples ', 'and oranges, ', 3)

print to_string example1
print to_string example2
print to_string example3
```

In the example, each time the calculate function is called, objects without related inheritance may be used (numbers, lists and strings). As long as the objects support the "+" and "*" methods, the operation will succeed. If translated to Ruby or Python, for example, the result of the code would be:

```
9
[1, 2, 3, 4, 5, 6, 1, 2, 3, 4, 5, 6]
apples and oranges, apples and oranges, apples and oranges,
```

Thus, duck typing allows polymorphism without inheritance. The only restriction that function calculate places on its variables is that they implement the "+" and the "*" methods.

The duck test can be seen in the following example (in Python). As far as the function in_the_forest is concerned, the Person object is a duck:

```
class Duck:
    def quack(self):
        print("Quaaaaaack!")
    def feathers(self):
        print("The duck has white and gray feathers.")

class Person:
    def quack(self):
        print("The person imitates a duck.")
    def feathers(self):
        print("The person takes a feather from the ground and shows it.")
    def name(self):
        print("John Smith")

def in_the_forest(duck):
    duck.quack()
    duck.feathers()

def game():
    donald = Duck()
    john = Person()
    in_the_forest(donald)
    in_the_forest(john)

game()
```

In statically typed languages

Certain usually statically typed languages such as Boo and the version 4 release of C# have extra type annotations^{[2][3]} that instruct the compiler to arrange for type checking of classes to occur at run-time rather than compile time, and include run-time type checking code in the compiled output. Such additions allow the language to enjoy most of the benefits of duck typing with the only drawback being the need to identify and specify such dynamic classes at compile time.

Comparison with other type systems

Structural type systems

Duck typing is similar to but distinct from structural typing. Structural typing is a static typing system that determines type compatibility and equivalence by a type's structure, whereas duck typing is dynamic and determines type compatibility by only that part of a type's structure that is accessed during run time.

The OCaml, Scala, and Go languages use structural type systems.

Protocols and Interfaces

Protocols and interfaces can provide some of the benefits of duck typing but duck typing is distinct in that no explicit interface is defined. For example, if a third party Java library implements a class you are not allowed to modify, you cannot use an instance of the class in place of an interface you have defined yourself, whereas duck typing would

allow this. Again, all of an interface must be satisfied for compatibility.

Templates or generic types

Template, or generic functions or methods apply the duck test in a static typing context; this brings all the advantages and disadvantages of static versus dynamic type checking in general. Duck typing can also be more flexible in that only the methods *actually called at run time* need to be implemented, while templates require implementation of all methods that *cannot be proven unreachable at compile time*.

Examples include the languages C++ and D with templates, which developed from Ada generics.

Criticism

One issue with duck typing is that it forces the programmer to have a much wider understanding of the code he or she is working with at any given time. In a strongly and statically typed language that uses type hierarchies and parameter type checking, it's much harder to supply an unexpected object type to a class. For instance, in Python, you could easily create a class called Wine, which expects a class implementing the "press" attribute as an ingredient. However, a class called Trousers might also implement the press() method. With Duck Typing, in order to prevent strange, hard-to-detect errors, the developer needs to be aware of each potential use of the method "press", even when it's conceptually unrelated to what he or she is working on.

In essence, the problem is that, "if it walks like a duck and quacks like a duck", it could be a dragon doing a duck impersonation. You may not always want to let dragons into a pond, even if they can impersonate a duck.

Proponents of duck typing, such as Guido van Rossum, argue that the issue is handled by testing, and the necessary knowledge of the codebase required to maintain it.^{[4][5]}

Criticisms around duck typing tend to be special cases of broader points of contention regarding dynamically typed versus statically typed programming language semantics.

History

Alex Martelli made an early (2000) use of the term in a message^[6] to the comp.lang.python newsgroup. He also highlighted misunderstanding of the literal duck test, which may indicate that the term was already in use:

In other words, don't check whether it IS-a duck: check whether it QUACKS-like-a duck, WALKS-like-a duck, etc, etc, depending on exactly what subset of duck-like behaviour you need to play your language-games with.

Implementations

In C#

In C# 4.0 the compiler and runtime collaborate to implement dynamic member lookup. Notice how parameter duck is declared *dynamic* in method InTheForest of class Program.

```
namespace DuckTyping {
    using System;

    public class Duck {
        public void Quack() {
            Console.WriteLine("Quaaaaaack!");
        }
    }
}
```

```
public void Feathers() {
    Console.WriteLine("The duck has white and gray feathers.");
}

public class Person {
    public void Quack() {
        Console.WriteLine("The person imitates a duck.");
    }

    public void Feathers() {
        Console.WriteLine("The person takes a feather from the ground and
shows it.");
    }
}

internal class Program {
    private static void InTheForest(dynamic duck) {
        duck.Quack();
        duck.Feathers();
    }

    private static void Game() {
        Duck donald = new Duck();
        Person john = new Person();
        InTheForest(donald);
        InTheForest(john);
    }

    private static void Main() {
        Game();
    }
}
```

In Cobra

In addition to static typing, Cobra allows one to declare objects of type 'dynamic' and send any message to them. At run-time the message passing will either succeed or throw an exception. The 'dynamic' type is the default for object variables and method arguments when a type has not been explicitly declared for them. This feature was inspired by Objective-C.^[7]

In ColdFusion

The web application scripting language ColdFusion allows function arguments to be specified as having type *any*. For this sort of argument, an arbitrary object can be passed in and method calls are bound dynamically at runtime. If an object does not implement a called method, a runtime exception is thrown which can be caught and handled gracefully. In ColdFusion 8, this can be picked up as a defined event `onMissingMethod()` rather than through an exception handler. An alternative argument type of `WEB-INF.cftags.component` restricts the passed argument to be a

ColdFusion Component (CFC), which provides better error messages should a non-object be passed in.

In Common Lisp

Common Lisp provides an object-oriented extension (Common Lisp Object System, or shorter CLOS). The combination of CLOS and Lisp's dynamic typing make duck typing a common programming style in Common Lisp.

With Common Lisp one also does not need to query the types, since at runtime an error will be signaled when a function is not applicable. The error can be handled with the Condition System of Common Lisp. Methods are defined outside of classes and can also be defined for specific objects.

```
;; We describe a protocol for 'duck-like' objects. Objects with methods
for
;; these three generic functions may be considered 'ducks', for all
intents
;; and purposes -- regardless of their superclass.
(defgeneric quack (something))
(defgeneric feathers (something))

;; Implementation of the protocol for class DUCK.
(defclass duck () ())

(defmethod quack ((a-duck duck))
  (print "Quaaaaaaack!"))

(defmethod feathers ((a-duck duck))
  (print "The duck has white and gray feathers."))

;; But we can also implement it for PERSON, without inheriting from
DUCK.
(defclass person () ())

(defmethod quack ((a-person person))
  (print "The person imitates a duck.))

(defmethod feathers ((a-person person))
  (print "The person takes a feather from the ground and shows it.))

;; IN-THE-FOREST does not need to be polymorphic. Its 'duck' argument
is
;; anything that implements the duck protocol above.
(defun in-the-forest (duck)
  (quack duck)
  (feathers duck))

;; GAME can also just be a regular function.
(defun game ()
  (let ((donald (make-instance 'duck))
        (john (make-instance 'person))))
```

```
(in-the-forest donald)
(in-the-forest john))

(game)
```

The usual development style of Common Lisp (by using a Lisp REPL like SLIME) allows also the interactive repair:

```
? (defclass cat () ())
#<STANDARD-CLASS CAT>
? (quack (make-instance 'cat))
> Error: There is no applicable method for the generic function:
>          #<STANDARD-GENERIC-FUNCTION QUACK #x300041C2371F>
>          when called with arguments:
>          (#<CAT #x300041C7EEFD>)
> If continued: Try calling it again
1 > (defmethod quack ((a-cat cat))
     (print "The cat imitates a duck."))
#<STANDARD-METHOD QUACK (CAT)>
1 > (continue)

"The cat imitates a duck."
```

This way software can be developed by extending partially working duck typed code.

In Java

In Java duck typing may be achieved with reflection.

```
import java.lang.reflect.InvocationHandler;
import java.lang.reflect.InvocationTargetException;
import java.lang.reflect.Method;
import java.lang.reflect.Proxy;

public class DuckTyping {

    interface Walkable { void walk(); }
    interface Swimmable { void swim(); }
    interface Quackable { void quack(); }

    public static void main(String[] args) {
        Duck d = new Duck();
        Person p = new Person();

        as(Walkable.class, d).walk();    //OK, duck has walk() method
        as(Swimmable.class, d).swim();   //OK, duck has swim() method
        as(Quackable.class, d).quack(); //OK, duck has quack() method

        as(Walkable.class, p).walk();    //OK, person has walk() method
        as(Swimmable.class, p).swim();   //OK, person has swim() method
    }

    private static void as(Class c, Object o) {
        Method m = null;
        try {
            m = c.getMethod("as");
        } catch (NoSuchMethodException e) {
            System.out.println("Method not found");
        }
        if (m != null) {
            m.invoke(o);
        }
    }
}
```

```
        as(Quackable.class, p).quack(); //Runtime Error, person does
not have quack() method
    }

    @SuppressWarnings("unchecked")
    static <T> T as(Class<T> t, final Object obj) {
        return (T) Proxy.newProxyInstance(t.getClassLoader(), new
Class[] {t},
            new InvocationHandler() {
                public Object invoke(Object proxy, Method method,
Object[] args) throws Throwable {
                    try {
                        return obj.getClass()
                            .getMethod(method.getName(),
method.getParameterTypes())
                            .invoke(obj, args);
                    } catch (NoSuchMethodException nsme) {
                        throw new NoSuchMethodError(nsme.getMessage());
                    } catch (InvocationTargetException ite) {
                        throw iteTargetException();
                    }
                }
            });
    }
}

class Duck {
    public void walk() {System.out.println("I'm Duck, I can
walk...");}
    public void swim() {System.out.println("I'm Duck, I can
swim...");}
    public void quack() {System.out.println("I'm Duck, I can
quack...");}
}

class Person {
    public void walk() {System.out.println("I'm Person, I can
walk...");}
    public void swim() {System.out.println("I'm Person, I can
swim...");}
    public void talk() {System.out.println("I'm Person, I can
talk...");}
}
```

In JavaScript

```

var Duck = function() {
    this.quack = function(){alert('Quaaaaaack!');};
    this.feathers = function(){alert('The duck has white and gray
feathers.');}
    return this;
};

var Person = function() {
    this.quack = function(){alert('The person imitates a duck.');}
    this.feathers = function(){alert('The person takes a feather from
the ground and shows it.');}
    this.name = function(){alert('John Smith');}
    return this;
};

var in_the_forest = function(duck) {
    duck.quack();
    duck.feathers();
};

var game = function() {
    var donald = new Duck();
    var john = new Person();
    in_the_forest(donald);
    in_the_forest(john);
};

game();

```

In Lua

Lua supports duck typing as part of the Metatable weak-typing system. Any reference to a table's member function is checked dynamically at run-time. If an object does not implement the requested function, a run-time error is produced. If a data member is requested but does not exist, a nil value is returned.

```

local duck_mt = { }
local duck_methods = { }
duck_mt.__index = duck_methods

function duck_methods:quack ( )
    print ( "Quaaaaaack!" )
end

function duck_methods:feathers ( )
    return "The duck has white and gray feathers."
end

```

```

function new_duck()
    return setmetatable ( { } , duck_mt )
end

local person_mt = { }
local person_methods = {}
person_mt.__index = person_methods

function person_methods:quack ( )
    print ( "The person imitates a duck." )
end

function person_methods:feathers ( )
    return "The person takes a feather from the ground and shows it."
end

function person_methods:get_name ( )
    return self.firstname .. " " .. self.lastname
end

function new_person ( t )
    return setmetatable ( t or {} , person_mt )
end

local function in_the_forest ( duck )
    duck:quack()
    print(duck:feathers())
end

local donald = new_duck()
local john = new_person { firstname="John", lastname="Smith" }
in_the_forest(donald)
in_the_forest(john)

```

In Objective-C

Objective-C, a cross between C and Smalltalk, allows one to declare objects of type 'id' and send any message to them (provided the method is declared somewhere), like in Smalltalk. The sender can test an object to see if it responds to a message, the object can decide at the time of the message whether it will respond to it or not, and if the sender sends a message a recipient cannot respond to, an exception is raised. Thus, duck typing is fully supported by Objective-C.

In Perl

Perl looks for method definitions in package set with `bless` function.

```
use strict;
```

```

package Duck;

sub hatch {
    bless \($self), shift;
}

sub quack {
    print "Quaaaaaack!\n";
}

sub feathers {
    print "The duck has white and gray feathers.\n";
}

package Person;

sub accept_birth {
    bless \($self), shift;
}

sub quack {
    print "The person imitates a duck.\n";
}

sub feathers {
    print "The person takes a feather from the ground and shows
it.\n";
}

package main;

sub in_the_forest
{
    my $duck = shift;
    $duck->quack();
    $duck->feathers();
}

my $duck = Duck->hatch();
my $person = Person->accept_birth();

in_the_forest( $duck );
in_the_forest( $person );

```

In PHP

```

<?php
class Duck{
    function quack(){ echo "Quack\n"; }
    function fly(){ echo "Flap, Flap\n"; }
}
class Person{

```

```

        function __construct ($name) {$this->name=$name;}
        function quack () {echo "{$this->name} walks in the forest and
imitates ducks to draw them\n";}
        function fly () { echo "{$this->name} takes an airplane\n";}
}

function QuackAndFly ($obj) {$obj->quack (); $obj->fly ();}

QuackAndFly (new Duck ());
QuackAndFly (new Person ("Jules Verne"));

/*
Output
.....
Quack
Flap, Flap

Jules Verne walks in the forest and imitates ducks to draw them.
Jules Verne takes an airplane.
*/

```

In PowerShell

This is the concept example from the beginning of the page.

```

Function calculate($a, $b, $c) {
    return ($a+$b)*$c
}

calculate 1 2 3
$(calculate (1, 2, 3) (4, 5, 6) 2)
calculate 'apples ' 'and oranges, ' 3

```

In Python

Duck typing is heavily used in Python. The Python's Glossary^[8] defines duck typing as follows:

Pythonic programming style that determines an object's type by inspection of its method or attribute signature rather than by explicit relationship to some type object ("If it looks like a duck and quacks like a duck, it must be a duck.") By emphasizing interfaces rather than specific types, well-designed code improves its flexibility by allowing polymorphic substitution. Duck-typing avoids tests using `type()` or `isinstance()`. Instead, it typically employs the *EAFP* (Easier to Ask Forgiveness than Permission) style of programming.

The canonical example of duck typing in Python is file-like classes. Classes can implement some or all of the methods of `file` and can be used where `file` would normally be used. For example, `GzipFile`^[9] implements a file-like object for accessing gzip-compressed data. `cStringIO`^[10] allows treating a Python string as a file. Sockets and files share many of the same methods as well. However, sockets lack the `tell()` method and cannot be used everywhere that `GzipFile` can be used. This shows the flexibility of duck typing: a file-like object can implement only methods it is able to, and consequently it can only be used in situations where it makes sense.

The EAFP principle describes the use of exception handling. For example instead of checking to see if some purportedly Duck-like object has a `quack()` method (using `if hasattr(mallard, "quack"):` ...) it's

usually preferable to wrap the attempted quacking with exception handling:

```
try:  
    mallard.quack()  
except (AttributeError, TypeError):  
    print("mallard can't quack()")
```

Advantages of this approach are that it encourages the structured handling of other classes of errors (so, for example, a mute Duck subclass could raise a "QuackException" which can be added to the wrapper without delving more deeply into the logic of the code, and it handles situations where different classes of objects might have naming collisions for incompatible members (for example, Mallard the purported medical professional might have a boolean attribute which classifies him as a "quack=True"; an attempt to perform Mallard.quack() would raise a TypeError)).

In the more practical examples of classes which implement file-like behavior the use of Python's exception handling facilities is generally preferred for handling a wide variety of I/O errors that can occur due to numerous environmental and operating system issues that are outside of the programmer's control. Here again the "duck typing" exceptions can be caught in their own clauses alongside the OS, I/O or other possible errors without complicated testing and error checking logic.

Simply stated: provided you can perform the job, we don't care who your parents are.

In Ruby

```
class Duck  
  def quack  
    puts "Quaaaaaaack!"  
  end  
  
  def feathers  
    puts "The duck has white and gray feathers."  
  end  
end  
  
class Person  
  def quack  
    puts "The person imitates a duck."  
  end  
  
  def feathers  
    puts "The person takes a feather from the ground and shows it."  
  end  
end  
  
def in_the_forest duck  
  duck.quack  
  duck.feathers  
end  
  
def game  
  donald = Duck.new  
  john = Person.new
```

```
in_the_forest donald
in_the_forest john
end

game
```

In Smalltalk

Duck typing is fundamental to Smalltalk. Variables have no data type, and can hold any object. Behavior is triggered by messages sent between objects. Any arbitrary string can be sent to any object as a message. The receiving object checks its method list for a matching behavior. This is the only approximation of type-checking in the language.

Moreover, a message with no matching method is not necessarily an error. In this case, the receiving object triggers its own `doesNotUnderstand:` method, inherited from `Object`. The default implementation raises an error, but this can be overridden to perform arbitrary operations based on the original message.

References

- [1] Heim, Michael (2007). *Exploring Indiana Highways* (<http://books.google.com/books?id=j7zds6xx7S0C&pg=PA68&dq=james+Riley>). Exploring America's Highway. pp. 68. ISBN 978-0-9744358-3-1..
- [2] Boo: Duck Typing (<http://boo.codehaus.org/Duck+Typing>)
- [3] Anders Hejlsberg Introduces C# 4.0 at PDC 2008 (<http://blogs.msdn.com/ericwhite/archive/2008/10/29/anders-hejlsberg-introduces-c-4-0-at-pdc-2008.aspx>)
- [4] Bruce Eckel. "Strong Typing vs. Strong Testing" (<http://www.mindviewinc.com/Blog/OldestIndex.php>). mindview..
- [5] Bill Venners. "Contracts in Python. A Conversation with Guido van Rossum, Part IV" (<http://www.artima.com/intv/pycontract.html>). Artima. .
- [6] <http://groups.google.com/group/comp.lang.python/msg/e230ca916be58835?hl=en&>
- [7] "Cobra - Acknowledgements" (<http://cobra-language.com/docs/acknowledgements/>). cobra-language.com. . Retrieved 2010-04-07.
- [8] <http://docs.python.org/glossary.html#term-duck-typing>
- [9] <http://docs.python.org/library/gzip.html#gzip.GzipFile>
- [10] <http://docs.python.org/library/stringio.html#module-cStringIO>

External links

- Duck Typing: Ruby (<http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/78502>)
- How to duck type? - the psychology of static typing in Ruby (<http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/100511>)
- Python documentation glossary entry on duck-typing (<http://docs.python.org/glossary.html#term-duck-typing>)
- Dr. Dobbs June 01 2005: "Templates and Duck Typing" (<http://www.ddj.com/cpp/184401971>)
- Javascript 'typeof' limitations and duck typing (<http://bigdingus.com/2007/12/08/just-what-is-this-javascript-object-you-handed-me/>)
- Python from a Java perspective - Part 2 - How duck typing influences class design and design principles (<http://blog.dhananjaynene.com/2008/09/python-from-java-how-duck-typing-influences-class-design-and-design-principles/>)
- Apache Software Foundation "commons" proxy project (<http://commons.apache.org/proxy>) provides DuckTyping implementation in Java

Ruby (programming language)

Ruby



Paradigm(s)	multi-paradigm: object-oriented, imperative, reflective, functional
Appeared in	1995
Designed by	Yukihiro Matsumoto
Developer	Yukihiro Matsumoto, et al.
Stable release	1.9.3-p286 (October 12, 2012)
Typing discipline	duck, dynamic
Scope	lexical, sometimes dynamic
Major implementations	Ruby MRI, YARV, Rubinius, MagLev JRuby, MacRuby, HotRuby, IronRuby
Influenced by	Ada, ^[1] C++, ^[1] CLU, ^[2] Dylan, ^[2] Eiffel, ^[1] Lisp, ^[2] Perl, ^[2] Python, ^[2] Smalltalk ^[2]
Influenced	Falcon, Fancy, ^[3] Groovy, Ioke, ^[4] Mirah, Nu, ^[5] Reia
OS	Cross-platform
License	Ruby License or BSD License ^{[6][7]}
Usual filename extensions	.rb, .rbw
Website	www.ruby-lang.org ^[8]
 Ruby Programming at Wikibooks	

Ruby is a dynamic, reflective, general-purpose object-oriented programming language that combines syntax inspired by Perl with Smalltalk-like features. It was also influenced by Eiffel and Lisp. Ruby was first designed and developed in the mid-1990s by Yukihiro "Matz" Matsumoto in Japan.

Ruby supports multiple programming paradigms, including functional, object oriented, imperative and reflective. It also has a dynamic type system and automatic memory management; it is therefore similar in varying respects to Smalltalk, Python, Perl, Lisp, Dylan, Pike, and CLU.

The standard 1.8.7 implementation is written in C, as a single-pass interpreted language. The language specifications for Ruby were developed by the Open Standards Promotion Center of the Information-Technology Promotion Agency (a Japanese government agency) for submission to the Japanese Industrial Standards Committee and then to the International Organization for Standardization. It was accepted as a Japanese Industrial Standard (JIS X 3017) in 2011^[9] and an international standard (ISO/IEC 30170) in 2012.^[10] As of 2010, there are a number of complete or upcoming alternative implementations of Ruby, including YARV, JRuby, Rubinius, IronRuby, MacRuby (and its iOS counterpart, RubyMotion), and HotRuby. Each takes a different approach, with IronRuby, JRuby, MacRuby and Rubinius providing just-in-time compilation and MacRuby also providing ahead-of-time compilation. The official

1.9 branch uses YARV, as will 2.0 (development), and will eventually supersede the slower Ruby MRI.

History

Ruby was conceived on February 24, 1993 by Yukihiro Matsumoto who wished to create a new language that balanced functional programming with imperative programming.^[11] Matsumoto has said, "I wanted a scripting language that was more powerful than Perl, and more object-oriented than Python. That's why I decided to design my own language."^[12]

At a Google Tech Talk in 2008 Matsumoto further stated, "I hope to see Ruby help every programmer in the world to be productive, and to enjoy programming, and to be happy. That is the primary purpose of Ruby language."^[13]

Choice of the name "Ruby"

The name "Ruby" was decided on during an online chat session between Matsumoto and Keiju Ishitsuka on February 24, 1993, before any code had been written for the language.^[14] Initially two names were proposed: "Coral" and "Ruby", with the latter being chosen by Matsumoto in a later email to Ishitsuka.^[15] Matsumoto has later remarked that a factor in choosing the name "Ruby" was because it was the birthstone of one of his colleagues.^{[16][17]}

First publication

The first public release of Ruby 0.95 was announced on Japanese domestic newsgroups on December 21, 1995.^{[18][19]} Subsequently three more versions of Ruby were released in two days.^[14] The release coincided with the launch of the Japanese-language *ruby-list* mailing list, which was the first mailing list for the new language.

Already present at this stage of development were many of the features familiar in later releases of Ruby, including object-oriented design, classes with inheritance, mixins, iterators, closures, exception handling and garbage collection.^[20]

Ruby 1.0

Ruby reached version 1.0 on December 25, 1996.^[14]

Following the release of Ruby 1.3 in 1999 the first English language mailing list ruby-talk began,^[12] which signalled a growing interest in the language outside of Japan. In September 2000, the first English language book Programming Ruby was printed, which was later freely released to the public, further widening the adoption of Ruby amongst English speakers.

Ruby 1.2

Ruby 1.2 was initially released in December 1998.

Ruby 1.4

Ruby 1.4 was initially released in August 1999.

Ruby 1.6

Ruby 1.6 was initially released in September 2000.

Ruby 1.8

Ruby 1.8 was initially released in August 2003, and was stable for a long time. Although deprecated, there is still code based on it. Ruby 1.8 is incompatible with Ruby 1.9.

Ruby on Rails

Around 2005, interest in the Ruby language surged in tandem with Ruby on Rails, a popular web application framework written in Ruby. Rails is frequently credited with making Ruby "famous".^[21]

Ruby 1.9

Ruby 1.9 was released in December 2007. The latest stable version of the reference implementation is 1.9.3 and is dual-licensed under the Ruby License and a BSD License. Adoption of 1.9 was slowed by changes from 1.8 which required many popular third party gems to be rewritten.

Ruby 1.9 introduces many significant changes over the 1.8 series.^[22] Examples:

- Block local variables (variables that are local to the block in which they are declared)
- An additional lambda syntax (`fun = ->(a,b) { puts a + b }`)
- Per-string character encodings are supported
- New socket API (IPv6 support)
- `require_relative` import security

Ruby 2.0

Ruby 1.9 will be followed by Ruby 2.0.^[23] Ruby 2.0 is rumored to be "100% compatible" with Ruby 1.9.3.^[24] As of October 2011, the plan is to have code freeze October 2012 and release February 2013.^[24]

Philosophy

Matsumoto has said that Ruby is designed for programmer productivity and fun, following the principles of good user interface design.^[25] He stresses that systems design needs to emphasize human, rather than computer, needs:^[26]

Often people, especially computer engineers, focus on the machines. They think, "By doing this, the machine will run faster. By doing this, the machine will run more effectively. By doing this, the machine will something something something." They are focusing on machines. But in fact we need to focus on humans, on how humans care about doing programming or operating the application of the machines. We are the masters. They are the slaves.

Ruby is said to follow the principle of least astonishment (POLA), meaning that the language should behave in such a way as to minimize confusion for experienced users. Matsumoto has said his primary design goal was to make a language which he himself enjoyed using, by minimizing programmer work and possible confusion. He has said that he had not applied the principle of least surprise to the design of Ruby,^[26] but nevertheless the phrase has come to be closely associated with the Ruby programming language. The phrase has itself been a source of surprise, as novice users may take it to mean that Ruby's behaviors try to closely match behaviors familiar from other languages. In a May 2005, discussion on the newsgroup `comp.lang.ruby`, Matsumoto attempted to distance Ruby from POLA, explaining that because any design choice will be surprising to someone, he uses a personal standard in evaluating surprise. If that personal standard remains consistent, there would be few surprises for those familiar with the standard.^[27]

Matsumoto defined it this way in an interview:^[26]

Everyone has an individual background. Someone may come from Python, someone else may come from Perl, and they may be surprised by different aspects of the language. Then they come up to me and say, 'I was surprised by this feature of the language, so Ruby violates the principle of least surprise.' Wait. Wait. The principle of least surprise is not for you only. The principle of least surprise means principle of least *my* surprise [*sic*]. And it means the principle of least surprise after you learn Ruby very well. For example, I was a C++ programmer before I started designing Ruby. I programmed in C++ exclusively for two or three years. And after two years of C++ programming, it still surprises me.

Features

- Thoroughly object-oriented with inheritance, mixins and metaclasses [28]
- Dynamic typing and duck typing
- Everything is an expression (even statements) and everything is executed imperatively (even declarations)
- Succinct and flexible syntax [29] that minimizes syntactic noise and serves as a foundation for domain-specific languages [30]
- Dynamic reflection and alteration of objects to facilitate metaprogramming [31]
- Lexical closures, iterators and generators, with a unique block syntax [32]
- Literal notation for arrays, hashes, regular expressions and symbols
- Embedding code in strings (interpolation)
- Default arguments
- Four levels of variable scope (global, class, instance, and local) denoted by sigils or the lack thereof
- Garbage collection
- First-class continuations
- Strict boolean coercion rules (everything is true except `false` and `nil`)
- Exception handling
- Operator overloading
- Built-in support for rational numbers, complex numbers and arbitrary-precision arithmetic
- Custom dispatch behavior (through `method_missing` and `const_missing`)
- Native threads and cooperative fibers
- Initial support for Unicode and multiple character encodings (still buggy as of version 1.9) [33]
- Native plug-in API in C
- Interactive Ruby Shell (a REPL)
- Centralized package management through RubyGems
- Implemented on all major platforms
- Large standard library

Semantics

Ruby is object-oriented: every value is an object, including classes and instances of types that many other languages designate as primitives (such as integers, booleans, and "null"). Variables always hold references to objects. Every function is a method and methods are always called on an object. Methods defined at the top level scope become members of the `Object` class. Since this class is an ancestor of every other class, such methods can be called on any object. They are also visible in all scopes, effectively serving as "global" procedures. Ruby supports inheritance with dynamic dispatch, mixins and singleton methods (belonging to, and defined for, a single instance rather than being defined on the class). Though Ruby does not support multiple inheritance, classes can import modules as mixins.

Ruby has been described as a multi-paradigm programming language: it allows procedural programming (defining functions/variables outside classes makes them part of the root, 'self' `Object`), with object orientation (everything is an object) or functional programming (it has anonymous functions, closures, and continuations; statements all have values, and functions return the last evaluation). It has support for introspection, reflection and metaprogramming, as well as support for interpreter-based^[34] threads. Ruby features dynamic typing, and supports parametric polymorphism.

According to the Ruby FAQ,^[35] "If you like Perl, you will like Ruby and be right at home with its syntax. If you like Smalltalk, you will like Ruby and be right at home with its semantics. If you like Python, you may or may not be put off by the huge difference in design philosophy between Python and Ruby/Perl."

Syntax

The syntax of Ruby is broadly similar to Perl and Python. Class and method definitions are signaled by keywords. In contrast to Perl, variables are not obligatorily prefixed with a sigil. When used, the sigil changes the semantics of scope of the variable. One difference from C and Perl is that keywords are typically used to define logical code blocks, without braces (i.e., pair of { and }). For practical purposes there is no distinction between expressions and statements.^[36] Line breaks are significant and taken as the end of a statement; a semicolon may be equivalently used. Unlike Python, indentation is not significant.

One of the differences of Ruby compared to Python and Perl is that Ruby keeps all of its instance variables completely private to the class and only exposes them through accessor methods (`attr_writer`, `attr_reader`, etc.). Unlike the "getter" and "setter" methods of other languages like C++ or Java, accessor methods in Ruby are created with a single line of code via metaprogramming. As invocation of these methods does not require the use of parentheses, it is trivial to change an instance variable into a full function, without modifying a single line of code or having to do any refactoring achieving similar functionality to C# and VB.NET property members.

Python's property descriptors are similar, but come with a tradeoff in the development process. If one begins in Python by using a publicly exposed instance variable, and later changes the implementation to use a private instance variable exposed through a property descriptor, code internal to the class may need to be adjusted to use the private variable rather than the public property. Ruby's design forces all instance variables to be private, but also provides a simple way to declare `set` and `get` methods. This is in keeping with the idea that in Ruby, one never directly accesses the internal members of a class from outside of it; rather, one passes a message to the class and receives a response.

See the *examples* section for samples of code demonstrating Ruby syntax.

Deviations from behavior elsewhere

Some features which differ notably from languages such as C or Perl:

- The language syntax is sensitive to the capitalization of identifiers, in most cases treating capitalized variables as constants.
- The sigils \$ and @ do not indicate variable data type as in Perl, but rather function as scope resolution operators.
- To denote a floating point without a decimal component, one must follow with a zero digit (99.0) or an explicit conversion (99.to_f). It is insufficient to append a dot (99.) since numbers are susceptible to method syntax.
- Boolean evaluation of non-boolean data is strict: 0, "" and [] are all evaluated to *true*. In C, the expression 0 ? 1 : 0 evaluates to 0 (i.e. false). In Ruby, however, it yields 1, as all numbers evaluate to true; only `nil` and `false` evaluate to *false*. A corollary to this rule is that Ruby methods by convention — for example, regular-expression searches — return numbers, strings, lists, or other non-false values on success, but `nil` on failure. This convention is also used in Smalltalk, where only the special objects `true` and `false` can be used in a boolean expression.
- Versions prior to 1.9 use plain integers to represent single characters, much like C. This may cause surprises when slicing strings: "abc"[0] yields 97 (the ASCII code of the first character in the string); to obtain "a" use "abc"[0,1] (a substring of length 1) or "abc"[0].chr.
- The notation `statement until expression`, like Perl but unlike other languages' equivalent statements (e.g. `do { statement } while (! (expression))`; in C/C++...), actually never runs the statement if the expression is already true. This is because `statement until expression` is actually syntactic sugar over `until expression; statement; end`, the equivalent of which in C/C++ is `while (! (expression)) { statement; }`, just as `statement if expression` is equivalent to `if (expression) { statement; }`. However, the notation `begin statement end until expression` in Ruby will in fact run the statement once even if the expression is already true, acting similar to

the "do-while" of other languages. (Matz has expressed a desire to remove the special behavior of `begin` statement `end` until `expression`,^[37] but it still exists as of Ruby 1.9.)

- Because constants are references to objects, changing what a constant refers to generates a warning, but modifying the object itself does not. For example, `Greeting << " world!" if Greeting == "Hello"` does not generate an error or warning. This is similar to final variables in Java or a const pointer to a non-const object in C++, but Ruby provides the functionality to "freeze" an object, unlike Java.

Some features which differ notably from other languages:

- The usual operators for conditional expressions, `and` and `or`, do not follow the normal rules of precedence: `and` does not bind tighter than `or`. Ruby also has expression operators `||` and `&&` which work as expected.

A list of so-called gotchas may be found in Hal Fulton's book *The Ruby Way*, 2nd ed (ISBN 0-672-32884-4), Section 1.5. A similar list in the 1st edition pertained to an older version of Ruby (version 1.6), some problems of which have been fixed in the meantime. `retry`, for example, now works with `while`, `until`, and `for`, as well as iterators.

Interaction

The Ruby official distribution also includes "`irb`", an interactive command-line interpreter which can be used to test code quickly. The following code fragment represents a sample session using `irb`:

```
$ irb
irb(main):001:0> puts "Hello, World"
Hello, World
=> nil
irb(main):002:0> 1+2
=> 3
```

Examples

The following examples can be run in a Ruby shell such as Interactive Ruby Shell or saved in a file and run from the command line by typing `ruby <filename>`.

Classic Hello world example:

```
puts "Hello World!"
```

Some basic Ruby code:

```
# Everything, including a literal, is an object, so this works:
-199.abs                                # 199
"ice is nice".length                      # 11
"ruby is cool.".index("u")                 # 1
"Nice Day Isn't It?".downcase.split("") .uniq.sort.join "# ''?acdeinsty"
```

Conversions:

```
puts "What's your favorite number?"
number = gets.chomp
output_number = number.to_i + 1
puts output_number.to_s + ' is a bigger and better favorite number.'
```

Strings

There are a variety of methods for defining strings in Ruby.

The following assignments are equivalent and support Variable interpolation:

```
a = "\nThis is a double-quoted string\n"
a = %Q{\nThis is a double-quoted string\n}
a = %{\nThis is a double-quoted string\n}
a = %/\nThis is a double-quoted string\n/
a = <<-BLOCK
```

```
This is a double-quoted string
BLOCK
```

The following assignments are equivalent and produce raw strings:

```
a = 'This is a single-quoted string'
a = %q{This is a single-quoted string}
```

Collections

Constructing and using an array:

```
a = [1, 'hi', 3.14, 1, 2, [4, 5]]

puts a[2]          # 3.14
puts a.[](2)       # 3.14
puts a.reverse     # [[4, 5], 2, 1, 3.14, 'hi', 1]
puts a.flatten.uniq # [1, 'hi', 3.14, 2, 4, 5]
```

Constructing and using an associative array (called hashes in Ruby):

```
hash = { :water => 'wet', :fire => 'hot' }
puts hash[:fire] # Prints: hot

hash.each_pair do |key, value| # Or: hash.each do |key, value|
  puts "#{key} is #{value}"
end

# Prints: water is wet
#         fire is hot

hash.delete :water # Deletes :water => 'wet'
hash.delete_if { |key,value| value=='hot' } # Deletes :fire => 'hot'
```

Blocks and iterators

The two syntaxes for creating a code block:

```
{ puts "Hello, World!" } # Note the { braces }
#or
do
  puts "Hello, World!"
end
```

When a code block is created it is always attached to a method as an optional block argument.

Parameter-passing a block to be a closure:

```
# In an object instance variable (denoted with '@'), remember a block.
def remember(&a_block)
  @block = a_block
end

# Invoke the above method, giving it a block which takes a name.
remember{|name| puts "Hello, #{name}!"}

# When the time is right (for the object) -- call the closure!
@block.call("Jon")
# => "Hello, Jon!"
```

Creating an anonymous function:

```
proc {|arg| print arg}
Proc.new {|arg| print arg}
lambda {|arg| print arg}

# introduced in Ruby 1.9
->(arg) {print arg}
```

Returning closures from a method:

```
def create_set_and_get(initial_value=0) # Note the default value of 0
  closure_value = initial_value
  return Proc.new {|x| closure_value = x}, Proc.new { closure_value }
end

setter, getter = create_set_and_get # ie. returns two values
setter.call(21)
getter.call # => 21

# You can also use a parameter variable as a binding for the closure.
# So the above can be rewritten as...

def create_set_and_get(closure_value=0)
  return proc {|x| closure_value = x} , proc { closure_value }
end
```

Yielding the flow of program control to a block which was provided at calling time:

```
def use_hello
  yield "hello"
end

# Invoke the above method, passing it a block.
use_hello {|string| puts string} # => 'hello'
```

Iterating over enumerations and arrays using blocks:

```
array = [1, 'hi', 3.14]
array.each {|item| puts item }
# => 1
# => 'hi'
# => 3.14

array.each_index {|index| puts "#{index}: #{array[index]} "}
# => 0: 1
# => 1: 'hi'
# => 2: 3.14

# The following uses a Range
(3..6).each {|num| puts num }
# => 3
# => 4
# => 5
# => 6
```

A method such as inject() can accept both a parameter and a block. Inject iterates over each member of a list, performing some function on it while retaining an aggregate. This is analogous to the foldl function in functional programming languages. For example:

```
[1,3,5].inject(10) {|sum, element| sum + element} # => 19
```

On the first pass, the block receives 10 (the argument to inject) as sum, and 1 (the first element of the array) as element; this returns 11. 11 then becomes sum on the next pass, which is added to 3 to get 14. 14 is then added to 5, to finally return 19.

Blocks work with many built-in methods:

```
File.open('file.txt', 'w') do |file| # 'w' denotes "write mode".
  file.puts 'Wrote some text.'
end                                # File is automatically closed here

File.readlines('file.txt').each do |line|
  puts line
end
# => Wrote some text.
```

Using an enumeration and a block to square the numbers 1 to 10 (using a range):

```
(1..10).collect {|x| x*x} # => [1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Classes

The following code defines a class named Person. In addition to 'initialize', the usual constructor to create new objects, it has two methods: one to override the `<=>` comparison operator (so `Array#sort` can sort by age) and the other to override the `to_s` method (so `Kernel#puts` can format its output). Here, "attr_accessor" is an example of metaprogramming in Ruby: "attr_accessor" defines getter and setter methods of instance variables, "attr_reader" only getter methods. Also, the last evaluated statement in a method is its return value, allowing the omission of an explicit 'return'.

```
class Person
  attr_reader :name, :age
  def initialize(name, age)
    @name, @age = name, age
  end
  def <=>(person) # Comparison operator for sorting
    age <=> person.age
  end
  def to_s
    "#{@name} (#{@age})"
  end
end

group = [
  Person.new("Bob", 33),
  Person.new("Chris", 16),
  Person.new("Ash", 23)
]

puts group.sort.reverse
```

The above prints three names in reverse age order:

```
Bob (33)
Ash (23)
Chris (16)
```

Open classes

In Ruby, classes are never closed: you can always add methods to an existing class. This applies to the classes you write as well as the standard, built-in classes. All you have to do is open up a class definition for an existing class, and the new contents you specify will be added to whatever's there. A simple example of adding a new method to the standard library's Time class:

```
# re-open Ruby's Time class
class Time
  def yesterday
    self - 86400
  end
end
```

```
today = Time.now # => Thu Aug 14 16:51:50 +1200 2008
yesterday = today.yesterday # => Wed Aug 13 16:51:50 +1200 2008
```

Adding methods to previously defined classes is often called monkey-patching. This practice, however, can lead to possible collisions of behavior and subsequent unexpected results, and is a concern for code scalability if performed recklessly.

Exceptions

An exception is raised with a `raise` call:

```
raise
```

An optional message can be added to the exception:

```
raise "This is a message"
```

You can also specify which type of exception you want to raise:

```
raise ArgumentError, "Illegal arguments!"
```

Alternatively, you can pass an exception instance to the `raise` method:

```
raise ArgumentError.new("Illegal arguments!")
```

This last construct is useful when you need to raise a custom exception class featuring a constructor which takes more than one argument:

```
class ParseError < Exception
  def initialize input, line, pos
    super "Could not parse '#{input}' at line #{line}, position #{pos}"
  end
end

raise ParseError.new("Foo", 3, 9)
```

Exceptions are handled by the `rescue` clause. Such a clause can catch exceptions which inherit from `StandardError`. Also supported for use with exceptions are `else` and `ensure`

```
begin
  # Do something
rescue
  # Handle exception
else
  # Do this if no exception was raised
ensure
  # Do this whether or not an exception was raised
end
```

It is a common mistake to attempt to catch all exceptions with a simple `rescue` clause. To catch all exceptions one must write:

```
begin
  # Do something
rescue Exception
```

```
# don't write just rescue -- that only catches StandardError, a
subclass of Exception
# Handle exception
end
```

Or catch particular exceptions:

```
begin
# ...
rescue RuntimeError
# handling
end
```

It is also possible to specify that the exception object be made available to the handler clause:

```
begin
# ...
rescue RuntimeError => e
# handling, possibly involving e, such as "print e.to_s"
end
```

Alternatively, the most recent exception is stored in the magic global `$!`.

You can also catch several exceptions:

```
begin
# ...
rescue RuntimeError, Timeout::Error => e
# handling, possibly involving e
end
```

Metaprogramming

Ruby code can programmatically modify, at runtime, aspects of its own structure that would be fixed in more rigid languages, such as class and method definitions. This sort of metaprogramming can be used to write more concise code and effectively extend the language.

For example, the following Ruby code generates new methods for the built-in String class, based on a list of colors. The methods wrap the contents of the string with an HTML tag styled with the respective color.

```
COLORS = { :black    => "000",
          :red      => "f00",
          :green    => "0f0",
          :yellow   => "ff0",
          :blue     => "00f",
          :magenta  => "f0f",
          :cyan     => "0ff",
          :white    => "fff" }

class String
COLORS.each do |color,code|
  define_method "in_#{color}" do
    "<span style=\"color: ##{code}\">\#{self}</span>"
```

```
  end
end
end
```

The generated methods could then be used like so:

```
"Hello, World!".in_blue
=> "<span style=\"color: #00f\>Hello, World!</span>"
```

To implement the equivalent in many other languages, the programmer would have to write each method (in_black, in_red, in_green, etc.) by hand.

Some other possible uses for Ruby metaprogramming include:

- intercepting and modifying method calls
- implementing new inheritance models
- dynamically generating classes from parameters
- automatic object serialization
- interactive help and debugging

More examples

More sample Ruby code is available as algorithms in the following articles:

- Exponentiating by squaring
- Trabb Pardo-Knuth algorithm

Implementations

The newest version of Ruby, the recently released version 1.9, has currently two working implementations:

- The official Ruby interpreter often referred to as the Matz's Ruby Interpreter or MRI. This implementation is written in C and uses its own Ruby-specific virtual machine,
- JRuby, a Java-based implementation that runs on the Java virtual machine

Ruby version 1.8 has another implementation called Rubinius, a reimplementation of Ruby focusing on writing as much of the core in Ruby as possible.

There are other less-known or upcoming implementations such as Cardinal (an implementation for the Parrot virtual machine), IronRuby (alpha version available since July 24, 2008),^[38] Ruby.NET^[39], XRuby^[40] and HotRuby (runs Ruby source code on a web browser and Flash).

The maturity of Ruby implementations tends to be measured by their ability to run the Ruby on Rails (Rails) framework, because it is a complex framework to implement, and it uses many Ruby-specific features. The point when a particular implementation achieves this goal is called *The Rails singularity*. The reference implementation (MRI), JRuby, and Rubinius^[41] are all able to run Rails unmodified in a production environment. IronRuby^{[42][43]} is starting to be able to run Rails test cases, but is still far from being production-ready.

Ruby is available on many operating systems, such as Linux, Mac OS X, Microsoft Windows, Windows Phone,^[44] Windows CE and most flavors of Unix.

Ruby 1.9 has recently been ported onto Symbian OS 9.x.^[45]

Repositories and libraries

The Ruby Application Archive (RAA), as well as RubyForge, serve as repositories for a wide range of Ruby applications and libraries, containing more than seven thousand items. Although the number of applications available does not match the volume of material available in the Perl or Python community, there are a wide range of tools and utilities which serve to foster further development in the language.

RubyGems has become the standard package manager for Ruby libraries. It is very similar in purpose to Perl's CPAN, although its usage is more like apt-get.

Recently, many new and existing libraries have found a home on GitHub, a service that offers version control repository hosting for Git.

References

- [1] Cooper, Peter (2009). *Beginning Ruby: From Novice to Professional*. Beginning from Novice to Professional (2nd ed.). Berkeley: APress. p. 101. ISBN 1-4302-2363-4. "To a lesser extent, Python, LISP, Eiffel, Ada, and C++ have also influenced Ruby."
- [2] Bini, Ola (2007). *Practical JRuby on Rails Web 2.0 Projects: Bringing Ruby on Rails to Java*. Berkeley: APress. p. 3. ISBN 1-59059-881-4. "It draws primarily on features from Perl, Smalltalk, Python, Lisp, Dylan, and CLU."
- [3] Bertels, Christopher (23 February 2011). "Introduction to Fancy" (<http://rubini.us/2011/02/23/introduction-to-fancy/>). *Rubinius blog*. Engine Yard. . Retrieved 2011-07-21.
- [4] Bini, Ola. "Ioke" (<http://ioke.org/>). *Ioke.org*. . Retrieved 2011-07-21. "inspired by Io, Smalltalk, Lisp and Ruby"
- [5] Burks, Tim. "About Nu™" (<http://programming.nu/about>). *Programming Nu™*. Neon Design Technology, Inc.. . Retrieved 2011-07-21.
- [6] COPYING in Ruby official source repository (<http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/COPYING?view=markup>)
- [7] BSDL in Ruby official source repository (<http://svn.ruby-lang.org/cgi-bin/viewvc.cgi/trunk/BSDL?view=markup>)
- [8] <http://www.ruby-lang.org/>
- [9] プログラム言語RubyのJIS規格（JIS X 3017）制定について (<http://www.ipa.go.jp/about/press/20110322.html>)
- [10] プログラム言語Ruby、国際規格として承認 (http://www.ipa.go.jp/about/press/20120402_2.html)
- [11] Ruby-Lang About Ruby (<http://www.ruby-lang.org/en/about/>)
- [12] <http://www.linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html> An Interview with the Creator of Ruby
- [13] <http://www.youtube.com/watch?v=oEkJvvGEtB4> Google Tech Talks - Ruby 1.9
- [14] <http://blog.nicksieger.com/articles/2006/10/20/rubyconf-history-of-ruby> History of Ruby
- [15] <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/88819> "[FYI: historic] The decisive moment of the language name Ruby.
(Re: [ANN] ruby 1.8.1)" - Email from Hiroshi Sugihara to ruby-talk
- [16] "The Ruby Language FAQ - 1.3 Why the name 'Ruby'?" (<http://www.ruby-doc.org/docs/ruby-doc-bundle/FAQ/FAQ.html>).
Ruby-Doc.org. . Retrieved April 10, 2012.
- [17] Yukihiko Matsumoto (June 11, 1999). "Re: the name of Ruby?" (<http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/394>).
Ruby-Talk mailing list. . Retrieved April 10, 2012.
- [18] More archeoleinguistics: unearthing proto-Ruby (<http://eigenclass.org/hiki/ruby+0.95>)
- [19] "Re: history of ruby" - Email from Yukihiko Matsumoto to ruby-talk (<http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/382>)
- [20] "TUTORIAL - ruby's features" - Email From Yukihiko Matsumoto to ruby-list (<http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-list/124>)
- [21] (<http://www.devarticles.com/c/a/Ruby-on-Rails/Web-Development-Ruby-on-Rails/>)
- [22] <http://slideshow.rubyforge.org/ruby19.html>
- [23] "Ruby 2.0 Implementation Work Begins: What is Ruby 2.0 and What's New?" (<http://www.rubyinside.com/ruby-2-0-implementation-work-begins-what-is-ruby-2-0-and-whats-new-5515.html>). . Retrieved 2011-12-23.
- [24] <http://www.rubyinside.com/ruby-2-0-release-schedule-announced-roll-on-2013-5536.html>
- [25] The Ruby Programming Language by Yukihiko Matsumoto on 2000-06-12 (informit.com) (<http://www.informit.com/articles/article.aspx?p=18225>)
- [26] The Philosophy of Ruby, A Conversation with Yukihiko Matsumoto, Part I by Bill Venners on 2003-09-29 (Artima Developer) (<http://www.artima.com/intv/ruby4.html>)
- [27] Ruby Weekly News 23rd - 29th May 2005 (<http://www.rubyweeklynews.org/20050529>)
- [28] An Interview with the Creator of Ruby (<http://linuxdevcenter.com/pub/a/linux/2001/11/29/ruby.html>)
- [29] Dynamic Productivity with Ruby (<http://www.artima.com/intv/tuesday3.html>)
- [30] martinfowler.com (<http://martinfowler.com/articles/languageWorkbench.html>)
- [31] Ruby - Add class methods at runtime (http://www.codeproject.com/useritems/Ruby_Dynamic_Methods.asp)
- [32] Blocks and Closures in Ruby (<http://www.artima.com/intv/closures.html>)
- [33] Unicode support in Ruby is too buggy compared to similar programming languages (<http://redmine.ruby-lang.org/issues/show/2034>)
- [34] Green threads

- [35] Ruby FAQ (<http://www.rootr.net/rubyfaq-2.html>)
- [36] *In Ruby's syntax, statement is just a special case of an expression which cannot appear as an argument (e.g. multiple assignment).* <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/1120>
statement [...] can not be part of expression unless grouped within parentheses. <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/2460>
- [37] <http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-core/6745>
- [38] Lam, John (2008-07-24). "IronRuby at OSCON" (<http://www.iunknow.com/2008/07/ironruby-at-oscon.html>). . Retrieved 2008-08-04.
"We're shipping our first binary release. In this package, we're taking a "batteries included" approach and shipping the Ruby standard libraries in it"
- [39] <http://rubydotnetcompiler.googlecode.com/>
- [40] <http://xruby.com/>
- [41] Peter Cooper (2010-05-18). "The Why, What, and How of Rubinius 1.0's Release" (<http://www.rubyinside.com/the-why-what-and-how-of-rubinius-1-0-s-release-3261.html>). .
- [42] John Lam (2008-05-25). "IronRuby / Rails Question" (<http://rubyforge.org/pipermail/ironruby-core/2008-May/001909.html>). . Retrieved 2008-05-25.
- [43] John Lam (2008-05-30). "IronRuby and Rails" (<http://www.iunknow.com/2008/05/ironruby-and-rails.html>). . Retrieved 2008-06-01.
- [44] "Iron Ruby on Windows Phone 7" (<http://msdn.microsoft.com/en-us/magazine/ff960707.aspx>). .
- [45] "Ruby 1.9 for Symbian OS" (<https://ella.pragmaticcomm.com/symbian-ruby>). . Retrieved 2010-07-28.

Further reading

- McAnally, Jeremy; Arkin, Assaf (March 28, 2009), *Ruby in Practice* (First ed.), Manning Publications, p. 360, ISBN 1-933988-47-9
- Thomas, Dave; Fowler, Chad; Hunt, Andy (April 28, 2009), *Programming Ruby 1.9: The Pragmatic Programmers' Guide* (<http://pragprog.com/titles/ruby3/programming-ruby-1-9>) (Third ed.), Pragmatic Bookshelf, p. 1000, ISBN 1-934356-08-5
- Flanagan, David; Matsumoto, Yukihiro (January 25, 2008), *The Ruby Programming Language* (<http://oreilly.com/catalog/9780596516178/>) (First ed.), O'Reilly Media, p. 446, ISBN 0-596-51617-7
- Baird, Kevin (June 8, 2007), *Ruby by Example: Concepts and Code* (<http://nostarch.com/ruby.htm>) (First ed.), No Starch Press, p. 326, ISBN 1-59327-148-4
- Fitzgerald, Michael (May 14, 2007), *Learning Ruby* (<http://oreilly.com/catalog/9780596529864>) (First ed.), O'Reilly Media, p. 255, ISBN 0-596-52986-4
- Cooper, Peter (March 26, 2007), *Beginning Ruby: From Novice to Professional* (<http://apress.com/book/view/9781590597668>) (First ed.), Apress, p. 664, ISBN 1-59059-766-4
- Fulton, Hal (November 4, 2006), *The Ruby Way* (<http://www.informit.com/store/product.aspx?isbn=0672328844>) (Second ed.), Addison-Wesley Professional, p. 888, ISBN 0-596-52369-6
- Carlson, Lucas; Richardson, Leonard (July 19, 2006), *Ruby Cookbook* (<http://oreilly.com/catalog/9780596523695/>) (First ed.), O'Reilly Media, p. 906, ISBN 0-596-52369-6

External links

- Official website (<http://www.ruby-lang.org/en/>)
- Ruby documentation site (<http://www.ruby-doc.org>)
- Ruby Draft Specification- Sep 2010 (<http://www.ipa.go.jp/osc/english/ruby/index.html>)
- Wiki: Ruby language and implementation specification (<http://spec.ruby-doc.org/>)
- Ruby (<http://www.dmoz.org/Computers/Programming/Languages/Ruby/>) at the Open Directory Project
- Try Ruby! (<http://tryruby.org/>) - web-based Ruby REPL
- The Great Ruby Shootout (December 2008) (<http://antoniocangiano.com/2008/12/09/the-great-ruby-shootout-december-2008/>): Ruby implementations comparison.
- Collingbourne, Huw (June 17 2006), The Little Book Of Ruby, free PDF eBook 1.1MB, pp. 87 (<http://www.sapphiresteel.com/The-Little-Book-Of-Ruby>)

- Collingbourne, Huw (April 18 2009), The Book Of Ruby, free PDF eBook 2.9MB, pp. 425 (<http://www.sapphiresteel.com/the-book-of-ruby>)
- Ruby.on-page.net (<http://ruby.on-page.net>) — simple Ruby manual with many samples
- Ruby User Guide (<http://www.rubyist.net/~slagell/ruby/index.html>) By Matz, the creator of Ruby. Translated into English.
- Ruby From Other Languages (<http://www.ruby-lang.org/en/documentation/ruby-from-other-languages/>)
- Writing C Extensions to Ruby (MRI 1.8) ([http://www.eqqon.com/index.php/Ruby_C_Extension_API_Documentation_\(Ruby_1.8\)](http://www.eqqon.com/index.php/Ruby_C_Extension_API_Documentation_(Ruby_1.8)))
- RubyFlow: Community Filtered Ruby News (<http://www.rubyflow.com/>)
- The Ruby Reflector (<http://rubyreflector.com>) Automated Ruby News
- Ruby Forum (<http://www.ruby-forum.com/forum/ruby>) - Gateway to the ruby-talk mailing list
- The Ruby in Stone (<http://www.dsprobotics.com/flowstone.html>) Ruby IDE inside FlowStone Programming Language

Be more, be complex...

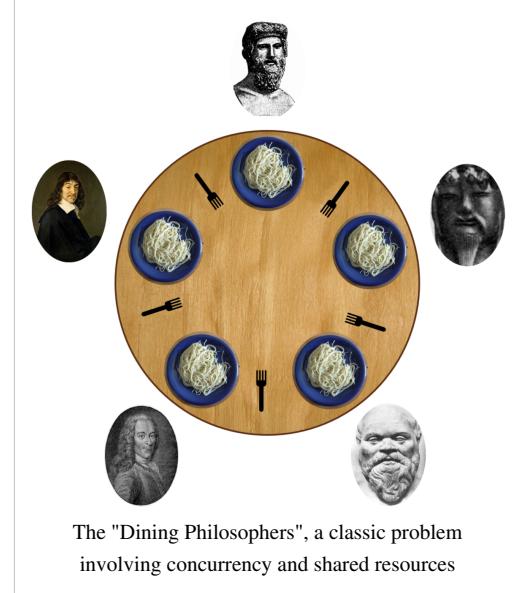
Concurrency (computer science)

In computer science, **concurrency** is a property of systems in which several computations are executing simultaneously, and potentially interacting with each other. The computations may be executing on multiple cores in the same chip, preemptively time-shared threads on the same processor, or executed on physically separated processors. A number of mathematical models have been developed for general concurrent computation including Petri nets, process calculi, the Parallel Random Access Machine model, the Actor model and the Reo Coordination Language.

Issues

Because computations in a concurrent system can interact with each other while they are executing, the number of possible execution paths in the system can be extremely large, and the resulting outcome can be indeterminate. Concurrent use of shared resources can be a source of indeterminacy leading to issues such as deadlock, and starvation.^[1]

The design of concurrent systems often entails finding reliable techniques for coordinating their execution, data exchange, memory allocation, and execution scheduling to minimize response time and maximise throughput.^[2]



The "Dining Philosophers", a classic problem involving concurrency and shared resources

Theory

Concurrency theory has been an active field of research in theoretical computer science. One of the first proposals was Carl Adam Petri's seminal work on Petri Nets in the early 1960s. In the years since, a wide variety of formalisms have been developed for modeling and reasoning about concurrency.

Models

A number of formalisms for modeling and understanding concurrent systems have been developed, including.^[3]

- The Parallel Random Access Machine^[4]
- The Actor model
- Computational bridging models such as the BSP model
- Petri nets
- Process calculi
- Tuple spaces, e.g., Linda
- SCOOP (Simple Concurrent Object-Oriented Programming)
- Reo Coordination Language

Some of these models of concurrency are primarily intended to support reasoning and specification, while others can be used through the entire development cycle, including design, implementation, proof, testing and simulation of

concurrent systems.

The proliferation of different models of concurrency has motivated some researchers to develop ways to unify these different theoretical models. For example, Lee and Sangiovanni-Vincentelli have demonstrated that a so-called "tagged-signal" model can be used to provide a common framework for defining the denotational semantics of a variety of different models of concurrency,^[5] while Nielsen, Sassone, and Winskel have demonstrated that category theory can be used to provide a similar unified understanding of different models.^[6]

The Concurrency Representation Theorem in the Actor model provides a fairly general way to represent concurrent systems that are closed in the sense that they do not receive communications from outside. (Other concurrency systems, e.g., process calculi can be modeled in the Actor model using a two-phase commit protocol.^[7]) The mathematical denotation denoted by a closed system S is constructed increasingly better approximations from an initial behavior called \perp_S using a behavior approximating function progression_S^i to construct a denotation (meaning) for S as follows:^[8]

$$\text{Denote } S \equiv \sqcup_{i \in \omega} \text{progression}_S^i(\perp_S)$$

In this way, S can be mathematically characterized in terms of all its possible behaviors.

Logics

Various types of temporal logic^[9] can be used to help reason about concurrent systems. Some of these logics, such as linear temporal logic and computational tree logic, allow assertions to be made about the sequences of states that a concurrent system can pass through. Others, such as action computational tree logic, Hennessy-Milner logic, and Lamport's temporal logic of actions, build their assertions from sequences of *actions* (changes in state). The principal application of these logics is in writing specifications for concurrent systems.^[1]

Practice

Concurrent programming encompasses the programming languages and algorithms used to implement concurrent systems. Concurrent programming is usually considered to be more general than parallel programming because it can involve arbitrary and dynamic patterns of communication and interaction, whereas parallel systems generally have a predefined and well-structured communications pattern. The base goals of concurrent programming include *correctness*, *performance* and *robustness*. Concurrent systems such as Operating systems and Database management systems are generally designed to operate indefinitely, including automatic recovery from failure, and not terminate unexpectedly (see Concurrency control). Some concurrent systems implement a form of transparent concurrency, in which concurrent computational entities may compete for and share a single resource, but the complexities of this competition and sharing are shielded from the programmer.

Because they use shared resources, concurrent systems in general require the inclusion of some kind of arbiter somewhere in their implementation (often in the underlying hardware), to control access to those resources. The use of arbiters introduces the possibility of indeterminacy in concurrent computation which has major implications for practice including correctness and performance. For example arbitration introduces unbounded nondeterminism which raises issues with model checking because it causes explosion in the state space and can even cause models to have an infinite number of states.

Some concurrent programming models include coprocesses and deterministic concurrency. In these models, threads of control explicitly yield their timeslices, either to the system or to another process.

References

- [1] Cleaveland, Rance; Scott Smolka (December, 1996). "Strategic Directions in Concurrency Research" (<http://doi.acm.org/10.1145/242223.242252>). *ACM Computing Surveys* **28** (4): 607. doi:10.1145/242223.242252. .
- [2] Campbell, Colin; Johnson, Ralph; Miller, Ade; Toub, Stephen (August 2010). *Parallel Programming with Microsoft .NET* (<http://msdn.microsoft.com/en-us/library/ff963542.aspx>). Microsoft Press. ISBN 978-0-7356-5159-3. .
- [3] Filman, Robert; Daniel Friedman (1984). *Coordinated Computing - Tools and Techniques for Distributed Software* (<http://ic.arc.nasa.gov/people/filman/text/dpl/dpl.html>). McGraw-Hill. ISBN 0-07-022439-0. .
- [4] Keller, Jörg; Christoph Keßler, Jesper Träff (2001). *Practical PRAM Programming*. John Wiley and Sons.
- [5] Lee, Edward; Alberto Sangiovanni-Vincentelli (December, 1998). "A Framework for Comparing Models of Computation". *IEEE Transactions on CAD* **17** (12): 1217–1229. doi:10.1109/43.736561.
- [6] Mogens Nielsen; Vladimiro Sassone and Glynn Winskel (1993). "Relationships Between Models of Concurrency" (<http://citeseer.ist.psu.edu/article/nielsen94relationships.html>). *REX School/Symposium*. .
- [7] Frederick Knabe. A Distributed Protocol for Channel-Based Communication with Choice PARLE 1992.
- [8] William Clinger (June 1981). *Foundations of Actor Semantics* (<https://dspace.mit.edu/handle/1721.1/6935>). Mathematics Doctoral Dissertation. MIT. .
- [9] Roscoe, Colin (2001). *Modal and Temporal Properties of Processes*. Springer. ISBN 0-387-98717-7.

Further reading

- Lynch, Nancy A. (1996). *Distributed Algorithms*. Morgan Kauffman. ISBN 1-55860-348-4.
- Tanenbaum, Andrew S.; Van Steen, Maarten (2002). *Distributed Systems: Principles and Paradigms*. Prentice Hall. ISBN 0-13-088893-1.
- Kurki-Suonio, Reino (2005). *A Practical Theory of Reactive Systems*. Springer. ISBN 3-540-23342-3.
- Garg, Vijay K. (2002). *Elements of Distributed Computing*. Wiley-IEEE Press. ISBN 0-471-03600-5.
- Magee, Jeff; Kramer, Jeff (2006). *Concurrency: State Models and Java Programming*. Wiley. ISBN 0-470-09355-2.

External links

- Concurrent Systems (<http://vl.fmnet.info/concurrent/>) at The WWW Virtual Library (<http://vlib.org/>)

Erlang (programming language)

Erlang



Paradigm(s)	multi-paradigm: concurrent, functional
Appeared in	1986
Designed by	Ericsson
Developer	Ericsson
Stable release	R15B02 ^[1] (5 September 2012)
Typing discipline	dynamic, strong
Major implementations	Erlang
Influenced by	Prolog, ML
Influenced	F#, Clojure, Rust, Scala, Opa, Reia
License	Modified MPL
Website	www.erlang.org ^[2]
Erlang Programming at Wikibooks	

Erlang ( /'ɜːrlæŋ/ *ER-lang*) is a general-purpose concurrent, garbage-collected programming language and runtime system. The sequential subset of Erlang is a functional language, with strict evaluation, single assignment, and dynamic typing. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. It supports hot swapping, so that code can be changed without stopping a system.^[3]

While threads require external library support in most languages, Erlang provides language-level features for creating and managing processes with the aim of simplifying concurrent programming. Though all concurrency is explicit in Erlang, processes communicate using message passing instead of shared variables, which removes the need for locks.

The first version was developed by Joe Armstrong in 1986.^[4] It was originally a proprietary language within Ericsson, but was released as open source in 1998.

History

The name "Erlang", attributed to Bjarne Däcker, has been understood as a reference to Danish mathematician and engineer Agner Krarup Erlang, and (initially at least) simultaneously as an abbreviation of "Ericsson Language".^{[4][5]}

Erlang was designed with the aim of improving the development of telephony applications. The initial version of Erlang was implemented in Prolog and was influenced by the programming language PLEX used in earlier Ericsson exchanges. According to Armstrong, the language went from lab product to real applications following the collapse of the next-generation AXE exchange named *AXE-N* in 1995. As a result, Erlang was chosen for the next ATM exchange *AXD*.^[4]

In 1998, the Ericsson AXD301 switch was announced, containing over a million lines of Erlang, and reported to achieve a reliability of nine "9"s.^[6] Shortly thereafter, Erlang was banned within Ericsson Radio Systems for new

products, citing a preference for non-proprietary languages. The ban caused Armstrong and others to leave Ericsson.^[7] The implementation was open sourced at the end of the year.^[4] The ban at Ericsson was eventually lifted, and Armstrong was re-hired by Ericsson in 2004.^[7]

In 2006, native symmetric multiprocessing support was added to the runtime system and virtual machine.^[4]

Philosophy

Quoting Mike Williams, one of the three inventors of Erlang:

1. Find the right methods—Design by Prototyping.
2. It is not good enough to have ideas, you must also be able to implement them and know they work.
3. Make mistakes on a small scale, not in a production project.

Functional programming examples

A factorial algorithm implemented in Erlang:

```
-module(fact).      % This is the file 'fact.erl', the module and the
filename must match
-export([fac/1]). % This exports the function 'fac' of arity 1 (1
parameter, no type, no name)

fac(0) -> 1; % If 0, then return 1, otherwise (note the semicolon ;
meaning 'else')
fac(N) when N > 0, is_integer(N) -> N * fac(N-1).
% Recursively determine, then return the result
% (note the period . meaning 'endif' or 'function end')
```

A sorting algorithm (similar to quicksort):

```
%% qsort:qsort(List)
%% Sort a list of items
-module(qsort).      % This is the file 'qsort.erl'
-export([qsort/1]). % A function 'qsort' with 1 parameter is exported
(no type, no name)

qsort([]) -> []; % If the list [] is empty, return an empty list
(nothing to sort)
qsort([Pivot|Rest]) ->
    % Compose recursively a list with 'Front' for all elements that
should be before 'Pivot'
    % then 'Pivot' then 'Back' for all elements that should be after
'Pivot'
    qsort([Front || Front <- Rest, Front < Pivot])
    ++ [Pivot] ++
    qsort([Back || Back <- Rest, Back >= Pivot]).
```

The above example recursively invokes the function `qsort` until nothing remains to be sorted. The expression `[Front || Front <- Rest, Front < Pivot]` is a list comprehension, meaning “Construct a list of elements `Front` such that `Front` is a member of `Rest`, and `Front` is less than `Pivot`.” `++` is the list concatenation operator.

A comparison function can be used for more complicated structures for the sake of readability.

The following code would sort lists according to length:

```
% This is file 'listsort.erl' (the compiler is made this way)
-module(listsort).

% Export 'by_length' with 1 parameter (don't care of the type and name)
-export([by_length/1]).

by_length(Lists) -> % Use 'qsort/2' and provides an anonymous function
as a parameter
    qsort(Lists, fun(A,B) -> A < B end).

qsort([], _) -> []; % If list is empty, return an empty list (ignore the
second parameter)
qsort([Pivot|Rest], Smaller) ->
    % Partition list with 'Smaller' elements in front of 'Pivot' and
not-'Smaller' elements
    % after 'Pivot' and sort the sublists.
    qsort([X || X <- Rest, Smaller(X,Pivot)], Smaller)
    ++ [Pivot] ++
    qsort([Y || Y <- Rest, not(Smaller(Y, Pivot))], Smaller).
```

Here again, a `Pivot` is taken from the first parameter given to `qsort()` and the rest of `Lists` is named `Rest`. Note that the expression

`[X || X <- Rest, Smaller(X,Pivot)]`

is no different in form from

`[Front || Front <- Rest, Front < Pivot]`

(in the previous example) except for the use of a comparison function in the last part, saying “Construct a list of elements `X` such that `X` is a member of `Rest`, and `Smaller` is true”, with `Smaller` being defined earlier as

`fun(A,B) -> A < B end`

Note also that the anonymous function is named `Smaller` in the parameter list of the second definition of `qsort` so that it can be referenced by that name within that function. It is not named in the first definition of `qsort`, which deals with the base case of an empty list and thus has no need of this function, let alone a name for it.

Data structures

Erlang has eight primitive data types:

1. **Integers:** integers are written as sequences of decimal digits, for example, 12, 12375 and -23427 are integers. Integer arithmetic is exact and only limited by available memory on the machine. (This is called arbitrary-precision arithmetic.)
2. **Atoms:** atoms are used within a program to denote distinguished values. They are written as strings of consecutive alphanumeric characters, the first character being a small letter. Atoms can contain any character if they are enclosed within single quotes and an escape convention exists which allows any character to be used within an atom.
3. **FLOATS:** floating point numbers use the IEEE 754 64-bit representation. (Range: $\pm 10^{308}$.)
4. **References:** references are globally unique symbols whose only property is that they can be compared for equality. They are created by evaluating the Erlang primitive `make_ref()`.

5. **Binaries:** a binary is a sequence of bytes. Binaries provide a space-efficient way of storing binary data. Erlang primitives exist for composing and decomposing binaries and for efficient input/output of binaries.
6. **Pids:** Pid is short for *process identifier*—a Pid is created by the Erlang primitive `spawn(...)`. Pids are references to Erlang processes.
7. **Ports:** ports are used to communicate with the external world. Ports are created with the built-in function (BIF) `open_port`. Messages can be sent to and received from ports, but these messages must obey the so-called "port protocol."
8. **Funs :** Funs are function closures. Funs are created by expressions of the form: `fun (...) -> ... end`.

And two compound data types:

1. **Tuples :** tuples are containers for a fixed number of Erlang data types. The syntax `{D1, D2, ..., Dn}` denotes a tuple whose arguments are `D1`, `D2`, ... `Dn`. The arguments can be primitive data types or compound data types. The elements of a tuple can be accessed in constant time.
2. **Lists :** lists are containers for a variable number of Erlang data types. The syntax `[D1 | D2 | ... | Dn | []]`. The first element of a list can be accessed in constant time. The first element of a list is called the *head* of the list. The remainder of a list when its head has been removed is called the *tail* of the list.

Two forms of syntactic sugar are provided:

1. **Strings :** strings are written as doubly quoted lists of characters, this is syntactic sugar for a list of the integer ASCII codes for the characters in the string, thus for example, the string "cat" is shorthand for `[99, 97, 116]`. It has partial support for unicode strings^[8]
2. **Records :** records provide a convenient way for associating a tag with each of the elements in a tuple. This allows us to refer to an element of a tuple by name and not by position. A pre-compiler takes the record definition and replaces it with the appropriate tuple reference.

Erlang has no method of defining classes, although there are external libraries available.^[9]

Concurrency and distribution orientation

Erlang's main strength is support for concurrency. It has a small but powerful set of primitives to create processes and communicate among them. Processes are the primary means to structure an Erlang application. Erlang processes loosely follow the communicating sequential processes (CSP) model. They are neither operating system processes nor operating system threads, but lightweight processes. Like operating system processes (but unlike operating system threads) they have no shared state between them. The estimated minimal overhead for each is 300 words,^[10] thus many of them can be created without degrading performance: a benchmark with 20 million processes has been successfully performed.^[11] Erlang has supported symmetric multiprocessing since release R11B of May 2006.

Inter-process communication works via a shared-nothing asynchronous message passing system: every process has a "mailbox", a queue of messages that have been sent by other processes and not yet consumed. A process uses the `receive` primitive to retrieve messages that match desired patterns. A message-handling routine tests messages in turn against each pattern, until one of them matches. When the message is consumed and removed from the mailbox the process resumes execution. A message may comprise any Erlang structure, including primitives (integers, floats, characters, atoms), tuples, lists, and functions.

The code example below shows the built-in support for distributed processes:

```
% Create a process and invoke the function web:start_server(Port,
MaxConnections)
ServerProcess = spawn(web, start_server, [Port, MaxConnections]),

% Create a remote process and invoke the function
% web:start_server(Port, MaxConnections) on machine RemoteNode
```

```

RemoteProcess = spawn(RemoteNode, web, start_server, [Port,
MaxConnections]),

    % Send a message to ServerProcess (asynchronously). The message
consists of a tuple
    % with the atom "pause" and the number "10".
ServerProcess ! {pause, 10},

    % Receive messages sent to this process
receive
    a_message -> do_something;
    {data, DataContent} -> handle(DataContent);
    {hello, Text} -> io:format("Got hello message: ~s", [Text]);
    {goodbye, Text} -> io:format("Got goodbye message: ~s", [Text])
end.

```

As the example shows, processes may be created on remote nodes, and communication with them is transparent in the sense that communication with remote processes works exactly as communication with local processes.

Concurrency supports the primary method of error-handling in Erlang. When a process crashes, it neatly exits and sends a message to the controlling process which can take action.^{[12][13]} This way of error handling increases maintainability and reduces complexity of code.

Implementation

The Ericsson Erlang implementation loads virtual machine bytecode which is converted to threaded code at load time. It also includes a native code compiler on most platforms, developed by the High Performance Erlang Project (HiPE) at Uppsala University. Since October 2001 the HiPE system is fully integrated in Ericsson's Open Source Erlang/OTP system.^[14] It also supports interpreting, directly from source code via abstract syntax tree, via script as of R11B-5 release of Erlang.

Hot code loading and modules

Erlang supports language-level Dynamic Software Updating. To implement this, code is loaded and managed as "module" units; the module is a compilation unit. The system can keep two versions of a module in memory at the same time, and processes can concurrently run code from each. The versions are referred to as the "new" and the "old" version. A process will not move into the new version until it makes an external call to its module.

An example of the mechanism of hot code loading:

```

%% A process whose only job is to keep a counter.

%% First version
-module(counter).

-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
receive
    {increment, Count} ->
        loop(Sum+Count);

```

```

{counter, Pid} ->
    Pid ! {counter, Sum},
    loop(Sum);
code_switch ->
    ?MODULE:codeswitch(Sum)
    % Force the use of 'codeswitch/1' from the latest MODULE
version
end.

codeswitch(Sum) -> loop(Sum).

```

For the second version, we add the possibility to reset the count to zero.

```

%% Second version
-module(counter).
-export([start/0, codeswitch/1]).

start() -> loop(0).

loop(Sum) ->
receive
    {increment, Count} ->
        loop(Sum+Count);
    reset ->
        loop(0);
    {counter, Pid} ->
        Pid ! {counter, Sum},
        loop(Sum);
    code_switch ->
        ?MODULE:codeswitch(Sum)
end.

codeswitch(Sum) -> loop(Sum).

```

Only when receiving a message consisting of the atom 'code_switch' will the loop execute an external call to codeswitch/1 (?MODULE is a preprocessor macro for the current module). If there is a new version of the "counter" module in memory, then its codeswitch/1 function will be called. The practice of having a specific entry-point into a new version allows the programmer to transform state to what is required in the newer version. In our example we keep the state as an integer.

In practice, systems are built up using design principles from the Open Telecom Platform which leads to more code upgradable designs. Successful hot code loading is a tricky subject; code needs to be written to make use of Erlang's facilities.

Distribution

In 1998, Ericsson released Erlang as open source to ensure its independence from a single vendor and to increase awareness of the language. Erlang, together with libraries and the real-time distributed database Mnesia, forms the Open Telecom Platform (OTP) collection of libraries. Ericsson and a few other companies offer commercial support for Erlang.

Since the open source release, Erlang has been used by several firms worldwide, including Nortel and T-Mobile.^[15] Although Erlang was designed to fill a niche and has remained an obscure language for most of its existence, its popularity is growing due to demand for concurrent services.^{[16][17]} Erlang has found some use in fielding MMORPG servers.^[18]

Erlang is available for many Unix-like operating systems, including Mac OS X, as well as Microsoft Windows.

Projects using Erlang

Projects using Erlang include:

- Database (distributed):
 - CouchDB, a document-based database that uses MapReduce
 - Couchbase (née Membase), database management system optimized for storing data behind interactive web applications.
 - Riak, a distributed database
 - SimpleDB, a distributed database that is part of Amazon Web Services^[19]
- Chat:
 - ejabberd, an Extensible Messaging and Presence Protocol (XMPP) instant messaging server
 - Facebook Chat system,^[20] based on ejabberd^[21]
 - Tuenti chat is based on ejabberd^[22]
- CMS:
 - Zotonic, a Content Management System and Web-Framework
- Queue:
 - RabbitMQ, an implementation of Advanced Message Queuing Protocol (AMQP)
- Desktop:
 - Wings 3D, a 3D subdivision modeller, used to model and texture polygon meshes.
- Web Servers:
 - Yaws web server.
- Tools
 - GitHub, a web-based hosting service for software development projects that use the Git (software) revision control system^[23]
- Mobile:
 - WhatsApp, mobile messenger^[24]
- Enterprise:
 - Issuu, an online digital publisher^[25]
 - Twitterfall, a service to view trends and patterns from Twitter^{[26][27]}
- Trading
 - Goldman Sachs, high-frequency trading programs.
 - Smarts, sports betting exchange

- Gaming
 - Battlestar Galactica Online game server by Bigpoint
 - Call of Duty server core^[28]

Companies using Erlang

Companies using Erlang in their production systems include:

- Amazon.com uses Erlang to implement SimpleDB, providing database services as a part of the Amazon Web Services offering.
- Yahoo! uses it in its social bookmarking service, Delicious, which has more than 5 million users and 150 million bookmarked URLs.
- Facebook uses Erlang to power the backend of its chat service, handling more than 100 million active users. It can be observed in some of its HTTP response headers:

```
Server: MochiWeb/1.0 (I'm not even supposed to be here today.)
```

- T-Mobile uses Erlang in its SMS and authentication systems.
- Motorola is using Erlang in call processing products in the public-safety industry.
- Ericsson uses Erlang in its support nodes, used in GPRS and 3G mobile networks worldwide.

References

- [1] R15B02 released (<http://www.erlang.org/news/34>)
- [2] <http://www.erlang.org>
- [3] Joe Armstrong, Bjarne Däcker, Thomas Lindgren, Håkan Millroth. "Open-source Erlang - White Paper" (http://ftp.sunet.se/pub/lang/erlang/white_paper.html). . Retrieved 31 July 2011.
- [4] Joe Armstrong, "History of Erlang", in *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, 2007, ISBN 978-1-59593-766-7
- [5] Erlang, the mathematician? (<http://www.erlang.org/pipermail/erlang-questions/1999-February/000098.html>)
- [6] "Concurrency Oriented Programming in Erlang" (<http://l12.ai.mit.edu/talks/armstrong.pdf>). 2 November 2002. .
- [7] "question about Erlang's future" (<http://erlang.org/pipermail/erlang-questions/2006-July/021336.html>). 6 July 2010. .
- [8] Unicode usage in Erlang official page (http://erlang.org/doc/apps/stdlib/unicode_usage.html)
- [9] Erlang Class Transformation project (<https://code.google.com/p/ect/>)
- [10] "Erlang Efficiency Guide - Processes" (http://www.erlang.org/doc/efficiency_guide/processes.html). .
- [11] Ulf Wiger (14 November 2005). "Stress-testing erlang" (<http://groups.google.com/group/comp.lang.functional/msg/33b7a62afb727a4f?dmode=source>). *comp.lang.functional.misc*. . Retrieved 25 August 2006.
- [12] Joe Armstrong. "Erlang robustness" (http://www.erlang.org/doc/getting_started/robustness.html). . Retrieved 15 July 2010.
- [13] "Erlang Supervision principles" (http://www.erlang.org/doc/design_principles/sup_princ.html). . Retrieved 15 July 2010.
- [14] "High Performance Erlang" (<http://www.it.uu.se/research/group/hipe/>). . Retrieved 26 March 2011.
- [15] "Who uses Erlang for product development?" (<http://www.erlang.org/faq/faq.html#AEN50>). *Frequently asked questions about Erlang*. . Retrieved 16 July 2007. "The largest user of Erlang is (surprise!) Ericsson. Ericsson use it to write software used in telecommunications systems. Many dozens projects have used it, a particularly large one is the extremely scalable AXD301 ATM switch. Other commercial users listed as part of the FAQ include: Nortel, Deutsche Flugsicherung (the German national air traffic control organisation), and T-Mobile."
- [16] "Programming Erlang" (http://www.ddj.com/linux-open-source/201001928?cid=RSSfeed_DDJ_OpenSource). . Retrieved 13 December 2008. "Virtually all language use shared state concurrency. This is very difficult and leads to terrible problems when you handle failure and scale up the system...Some pretty fast-moving startups in the financial world have latched onto Erlang; for example, the Swedish www.kreditor.se."
- [17] "Erlang, the next Java" (<http://www.cincomsmalltalk.com/userblogs/ralph/blogView?showComments=true&entry=3364027251>). . Retrieved 8 October 2008. "I do not believe that other languages can catch up with Erlang anytime soon. It will be easy for them to add language features to be like Erlang. It will take a long time for them to build such a high-quality VM and the mature libraries for concurrency and reliability. So, Erlang is poised for success. If you want to build a multicore application in the next few years, you should look at Erlang."
- [18] Clarke, Gavin (5 Feb 2011). "Battlestar Galactica vets needed for online roleplay" (http://www.theregister.co.uk/2011/02/05/battlestar_galactica_mmp/). *Music and Media*. The Reg. . Retrieved 8 February 2011.
- [19] What You Need To Know About Amazon SimpleDB (<http://www.satine.org/archives/2007/12/13/amazon-simpdb/>)
- [20] http://www.facebook.com/note.php?note_id=16787213919&id=9445547199&index=2

- [21] <http://developers.facebook.com/news.php?blog=1&story=110>
- [22] <http://blog.tuenti.com/dev/chat-in-the-making/>
- [23] <http://www.infoq.com/interviews/erlang-and-github>
- [24] <http://blog.whatsapp.com/index.php/2012/01/1-million-is-so-2011/>
- [25] <http://www.version2.dk/artikel/saadan-fik-dansk-sukses-website-held-med-erlang-og-amazon-11898>
- [26] <http://twitter.com/jalada/status/1206606823>
- [27] <http://twitter.com/jalada/statuses/1234217518>
- [28] "Erlang and First-Person Shooters" (<http://www.erlang-factory.com/upload/presentations/395/ErlangandFirst-PersonShooters.pdf>). . Retrieved 9 August 2012. "Presentation about Erlang and Call of Duty from Demonware."

Further reading

- Joe Armstrong (2003). *Making reliable distributed systems in the presence of software errors* (http://www.sics.se/~joe/thesis/armstrong_thesis_2003.pdf). Ph.D. Dissertation. The Royal Institute of Technology, Stockholm, Sweden.
- Armstrong, J. (2007). "A history of Erlang". *Proceedings of the third ACM SIGPLAN conference on History of programming languages - HOPL III*. pp. 6–1. doi:10.1145/1238844.1238850. ISBN 978159593766X.
- Early history of Erlang (<http://www.erlang.se/publications/bjarnelic.pdf>) by Bjarne Däcker
- "Mnesia - A distributed robust DBMS for telecommunications applications". *First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*: 152–163. 1999.
- Armstrong, Joe; Virding, Robert; Williams, Mike; Wikstrom, Claes (16 January 1996). *Concurrent Programming in Erlang* (http://www.erlang.org/erlang_book_toc.html) (2nd ed.). Prentice Hall. p. 358. ISBN 978-0-13-508301-7.
- Armstrong, Joe (11 July 2007). *Programming Erlang: Software for a Concurrent World* (<http://pragprog.com/titles/jaerlang/programming-erlang>) (1st ed.). Pragmatic Bookshelf. p. 536. ISBN 978-1-934356-00-5.
- Thompson, Simon J.; Cesarini, Francesco (19 June 2009). *Erlang Programming: A Concurrent Approach to Software Development* (<http://www.erlangprogramming.org>) (1st ed.). Sebastopol, California: O'Reilly Media, Inc. p. 496. ISBN 978-0-596-51818-9.
- Cant, Geoff (1 March 2010). *Mastering Erlang: Writing Real World Applications* (<http://www.apress.com/book/view/9781430227694>) (1st ed.). Apress. p. 350. ISBN 978-1-4302-2769-4.
- Logan, Martin; Merritt, Eric; Carlsson, Richard (28 May 2010). *Erlang and OTP in Action* (1st ed.). Greenwich, CT: Manning Publications. p. 500. ISBN 978-1-933988-78-8.
- Gerakines, Nick (2 August 2010). *Erlang Web Applications: Problem - Design - Solutions* (<http://eu.wiley.com/WileyCDA/WileyTitle/productCd-0470743840.html>) (1st ed.). John Wiley & Sons. p. 512. ISBN 978-0-470-74384-3.
- Martin, Brown (10 May 2011). "Introduction to programming in Erlang, Part 1: The basics" (<http://www.ibm.com/developerworksopensource/library/os-erlang1/index.html>). *developerWorks*. IBM. Retrieved 10 May 2011.
- Martin, Brown (17 May 2011). "Introduction to programming in Erlang, Part 2: Use advanced features and functionality" (<http://www.ibm.com/developerworksopensource/library/os-erlang2/index.html>). *developerWorks*. IBM. Retrieved 17 May 2011.

External links

- Official website (<http://www.erlang.org>)
- Erlang (<http://www.dmoz.org/Computers/Programming/Languages/Erlang/>) at the Open Directory Project
- trapexit.org (<http://trapexit.org/>), site with much Erlang/OTP information
- Erlang: The Movie (<http://www.archive.org/details/ErlangTheMovie>)
- Learn You Some Erlang (<http://www.learnyousomeerlang.com/>), tutorial for beginners
- erldocs.com (<http://erldocs.com/>), alternative topic documentation
- Joe Armstrong on Erlang (<http://www.se-radio.net/2008/03/episode-89-joe-armstrong-on-erlang>), Software Engineering Radio Podcast

New Guys

D (programming language)

D programming language

Paradigm(s)	multi-paradigm: imperative, object-oriented, functional, meta
Appeared in	2001 ^[1]
Designed by	Walter Bright, Andrei Alexandrescu (since 2006)
Developer	Digital Mars, Andrei Alexandrescu (since 2006)
Stable release	2.060 / 1.075 ^{[2][3]} (August 2, 2012) ^[4]
Typing discipline	strong, static
Major implementations	DMD (reference implementation), GDC, LDC
Influenced by	C, C++, C#, Eiffel, Java, Python, Ruby
Influenced	MiniD, DScript, Vala, Qore
OS	DMD: Unix-like, Windows, Mac OS X
License	GPL/Artistic (DMD frontend), Boost (standard and runtime libraries), source available (DMD backend), ^[5] Fully open-source (LDC and GDC) ^[6]
Usual filename extensions	.d
Website	[dlang.org dlang.org]
 D Programming at Wikibooks	

The **D programming language** is an object-oriented, imperative, multi-paradigm system programming language created by Walter Bright of Digital Mars. Though it originated as a re-engineering of C++, D is a distinct language, having redesigned some core C++ features while also taking inspiration from other languages, notably Java, Python, Ruby, C#, and Eiffel.

D's design goals attempt to combine the performance of compiled languages with the safety and expressive power of modern dynamic languages. Idiomatic D code is commonly as fast as equivalent C++ code, while being shorter and memory-safe. Type inference, automatic memory management and syntactic sugar for common types allow faster development, while bounds checking, design by contract features and a concurrency-aware type system help reduce the occurrence of bugs.^[7]

Features

D is designed with lessons learned from practical C++ usage rather than from a theoretical perspective. Even though it uses many C/C++ concepts it also discards some, and as such is not compatible with C/C++ source code. It adds to the functionality of C++ by also implementing design by contract, unit testing, true modules, garbage collection, first class arrays, associative arrays, dynamic arrays, array slicing, nested functions, inner classes, closures, anonymous functions, compile time function execution, lazy evaluation and has a reengineered template syntax. D retains C++'s ability to do low-level coding, and adds to it with support for an integrated inline assembler. C++ multiple inheritance is replaced by Java style single inheritance with interfaces and mixins. D's declaration, statement and expression syntax closely matches that of C++.

The inline assembler typifies the differences between D and application languages like Java and C#. An inline assembler lets programmers enter machine-specific assembly code within standard D code, a method often used by system programmers to access the low-level features of the processor needed to run programs that interface directly with the underlying hardware, such as operating systems and device drivers.

D has built-in support for documentation comments, allowing automatic documentation generation.

Programming paradigms

D supports five main programming paradigms—imperative, object-oriented, metaprogramming, functional and concurrent (Actor model).

Imperative

Imperative programming in D is almost identical to C. Functions, data, statements, declarations and expressions work just as in C, and the C runtime library can be accessed directly. Some notable differences between D and C in the area of imperative programming include D's `foreach` loop construct, which allows looping over a collection, and nested functions, which are functions that are declared inside of another and may access the enclosing function's local variables.

Object-oriented

Object-oriented programming in D is based on a single inheritance hierarchy, with all classes derived from class `Object`. D does not support multiple inheritance; instead, it uses Java-style interfaces, which are comparable to C++ pure abstract classes, and mixins, which allow separating common functionality out of the inheritance hierarchy. D also allows declaring static and final (non-virtual) methods in interfaces.

Metaprogramming

Metaprogramming is supported by a combination of templates, compile time function execution, tuples, and string mixins. The following examples demonstrate some of D's compile-time features.

Templates in D can be written in a more imperative style compared to C++ functional style for templates. This is a regular function that calculates the factorial of a number:

```
ulong factorial(ulong n)
{
    if(n < 2)
        return 1;
    else
        return n * factorial(n - 1);
}
```

Here, the use of `static if`, D's compile-time conditional construct, is demonstrated to construct a template that performs the same calculation using code that is similar to that of the above function:

```
template Factorial(ulong n)
{
    static if(n < 2)
        const Factorial = 1;
    else
        const Factorial = n * Factorial!(n - 1);
}
```

In the following two examples, the template and function defined above are used to compute factorials. The types of constants need not be specified explicitly as the compiler infers their types from the right-hand sides of assignments:

```
const fact_7 = Factorial!(7);
```

This is an example of compile time function execution. Ordinary functions may be used in constant, compile-time expressions provided they meet certain criteria:

```
const fact_9 = factorial(9);
```

The `std.metastrings.Format` template performs `printf`-like data formatting, and the "msg" pragma displays the result at compile time:

```
import std.metastrings;
pragma(msg, Format!("7! = %s", fact_7));
pragma(msg, Format!("9! = %s", fact_9));
```

String mixins, combined with compile-time function execution, allow generating D code using string operations at compile time. This can be used to parse domain-specific languages to D code, which will be compiled as part of the program:

```
import FooToD; // hypothetical module which contains a function that
parses Foo source code
                    // and returns equivalent D code
void main()
{
    mixin(fooToD(import("example.foo")));
}
```

Functional

```
import std.stdio, std.algorithm, std.range;

void main()
{
    immutable int[] a1 = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
    immutable int[] a2 = [6, 7, 8, 9];

    // must be immutable to allow access from inside mySum
    immutable pivot = 5;

    int mySum(in int a, in int b) pure nothrow // pure function
```

```
{  
    if (b <= pivot) // ref to enclosing-scope  
        return a + b;  
    else  
        return a;  
}  
  
// passing a delegate (closure)  
immutable result = reduce!mySum(chain(a1, a2));  
writeln("Result: ", result); // output is "15"  
}
```

Concurrent

```
import std.stdio, std.concurrency;  
  
void foo()  
{  
    bool cont = true;  
  
    while (cont)  
    {  
        receive( // delegates are used to match the message type  
            (int msg) => writeln("int received: ", msg),  
            (Tid sender) { cont = false; sender.send(-1); },  
            (Variant v) => writeln("huh?") // Variant matches any type  
        );  
    }  
}  
  
void main()  
{  
    auto tid = spawn(&foo); // spawn a new thread running foo()  
  
    foreach (i; 0 .. 10)  
        tid.send(i); // send some integers  
    tid.send(1.0f); // send a float  
    tid.send("hello"); // send a string  
    tid.send(thisTid); // send a struct (Tid)  
  
    receive((int x) => writeln("Main thread received message: ", x));  
}
```

Memory management

Memory is usually managed with garbage collection, but specific objects can be finalized immediately when they go out of scope. Explicit memory management is possible using the overloaded operators `new` and `delete`, and by simply calling C's `malloc` and `free` directly. Garbage collection can be controlled: programmers can add and exclude memory ranges from being observed by the collector, can disable and enable the collector and force a generational or a full collection cycle.^[8] The manual gives many examples of how to implement different highly optimized memory management schemes for when garbage collection is inadequate in a program.^[9]

Interaction with other systems

C's application binary interface (ABI) is supported as well as all of C's fundamental and derived types, enabling direct access to existing C code and libraries. D bindings are available for many popular C libraries. C's standard library is part of standard D.

Because C++ does not have a single standard ABI, D can only fully access C++ code that is written to the C ABI. The D parser understands an `extern (C++)` calling convention for limited linking to C++ objects.

On Microsoft Windows, D can access Component Object Model (COM) code.

History

Walter Bright decided to start working on a new language in 1999. D was first released in December 2001,^[1] and reached version 1.0 in January 2007.^[10] The first version of the language (D1) concentrated on the imperative, object oriented and metaprogramming paradigms,^[11] similar to C++.

Dissatisfied with Phobos, D's official runtime and standard library, members of the D community created an alternative runtime and standard library named Tango. The first public Tango announcement came within days of D 1.0's release.^[12] Tango adopted a different programming style, embracing OOP and high modularity. Being a community-led project, Tango was more open to contributions, which allowed it to progress faster than the official standard library. At that time, Tango and Phobos were incompatible due to different runtime support APIs (the garbage collector, threading support, etc.). This made it impossible to use both libraries in the same project. The existence of two libraries, both widely in use, has led to significant dispute due to some packages using Phobos and others using Tango.^[13]

In June 2007, the first version of D2 was released.^[2] The beginning of D2's development signalled the stabilization of D1; the first version of the language has since been in maintenance, only receiving corrections and implementation bugfixes. D2 was to introduce breaking changes to the language, beginning with its first experimental `const` system. D2 later added numerous other language features, such as closures, purity, and support for the functional and concurrent programming paradigms. D2 also solved standard library problems by separating the runtime from the standard library. The completion of a D2 Tango port was announced in February 2012.^[14]

The release of Andrei Alexandrescu's book *The D Programming Language* on June 12, 2010 marked the stabilization of D2, which today is commonly referred to as just "D".

In January 2011, D development moved from a bugtracker / patch-submission basis to GitHub. This has led to a significant increase in contributions to the compiler, runtime and standard library.^[15]

Implementations

Most current D implementations compile directly into machine code for efficient execution.

- *DMD* — The Digital Mars D compiler is the official D compiler by Walter Bright. The compiler front-end is licensed under both the Artistic License and the GNU GPL; the source code for the front-end is distributed with the compiler binaries. The compiler back-end source code is available but not under an open source license.^[5]
- *GDC* — A front-end for the GCC back-end, built using the open DMD compiler source code.^[16]
- *LDC* — A compiler based on the DMD front-end that uses Low Level Virtual Machine (LLVM) as its compiler back-end. The first release-quality version was published on January 9, 2009.^[17] It supports both versions: 1.0 and version 2.0.^[18]
- *D Compiler for .NET* — A back-end for the D programming language 2.0 compiler.^{[19][20]} It compiles the code to Common Intermediate Language (CIL) bytecode rather than to machine code. The CIL can then be run via a Common Language Infrastructure (CLR) virtual machine.

Development tools

Editors and integrated development environments (IDEs) supporting D include Eclipse, Microsoft Visual Studio, SlickEdit, Emacs, vim, SciTE, Smultron, TextMate, MonoDevelop, Zeus,^[21] and Geany among others.^[22]

- Eclipse plug-ins for D include: DDT,^[23] and Descent (dead project).^[24]
- Visual Studio integration is provided by VisualID.^[25]
- Vim supports both syntax highlighting and code completion (through patched Ctags).
- A bundle is available for TextMate, and the Code::Blocks IDE includes partial support for the language. However, standard IDE features such as code completion or refactoring are not yet available, though they do work partially in Code::Blocks (due to D's similarity to C).
- A plugin for Xcode 3 is available, D for Xcode, to enable D-based projects and development.^[26]
- An AddIn for MonoDevelop is available, named Mono-D.^[27]

Open source D IDEs for Windows exist, some written in D, such as Poseidon,^[28] D-IDE,^[29] and Entice Designer.^[30]

D applications can be debugged using any C/C++ debugger, like GDB or WinDbg, although support for various D-specific language features is extremely limited. On Windows, D programs can be debugged using Ddbg^[31], or Microsoft debugging tools (WinDBG and Visual Studio), after having converted the debug information using cv2pdb^[32]. The ZeroBUGS^[33] debugger for Linux has experimental support for the D language. Ddbg can be used with various IDEs or from the command line; ZeroBUGS has its own graphical user interface (GUI).

Examples

Example 1

This example program prints its command line arguments. The `main` function is the entry point of a D program, and `args` is an array of strings representing the command line arguments. A `string` in D is an array of characters, represented by `char[]` in D1, or `immutable(char)[]` in D2.

```
import std.stdio: writeln;

void main(string[] args)
{
    foreach (i, arg; args)
        writeln("args[%d] = '%s'", i, arg);
}
```

The `foreach` statement can iterate over any collection. In this case, it is producing a sequence of indexes (`i`) and values (`arg`) from the array `args`. The index `i` and the value `arg` have their types inferred from the type of the array `args`.

Example 2

The following shows several D capabilities and D design trade-offs in a very short program. It iterates the lines of a text file named `words.txt` that contains a different word on each line, and prints all the words that are anagrams of other words.

```
import std.stdio, std.algorithm, std.range, std.string;

void main()
{
    dstring[][][dstring] signs2words;
    foreach(dchar[] w; lines(File("words.txt")))
    {
        w = w.chomp().toLowerCase();
        immutable key = w.dup.sort().release().idup;
        signs2words[key] ~= w.idup;
    }

    foreach(words; signs2words)
        if(words.length > 1)
            writeln(words.join(" "));
}
```

1. `signs2words` is a built-in associative array that maps `dstring` (32-bit / char) keys to arrays of `dstrings`. It is similar to `defaultdict(list)` in Python.
2. `lines(File())` yields lines lazily, with the newline. It has to then be copied with `idup` to obtain a string to be used for the associative array values (the `idup` property of arrays returns an immutable duplicate of the array, which is required since the `dstring` type is actually `immutable(dchar[])`). Built-in associative arrays require immutable keys.
3. The `~` operator appends a new `dstring` to the values of the associate dynamic array.
4. `toLowerCase`, `join` and `chomp` are string functions that D allows to use with a method syntax. The name of such functions is often very similar to Python string methods. The `toLowerCase` converts a string to lower case, `join(" ")` joins an array of strings into a single string using a single space as separator, and `chomp` removes the eventually present newline from the end of the string.
5. The `sort` is an `std.algorithm` function that sorts the array in place, creating a unique signature for words that are anagrams of each other. The `release()` method of `sort()` is handy to keep the code as a single expression.
6. The second `foreach` iterates on the values of the associative array, it's able to infer the type of `words`.
7. `key` is assigned to an immutable variable, its type is inferred.
8. UTF-32 `dchar[]` is used instead of normal UTF-8 `char[]` otherwise `sort()` refuses to sort it. There are more efficient ways to write this program, that use just UTF-8.

References

- [1] "D Change Log to Nov 7 2005" (<http://www.digitalmars.com/d/1.0/changelog1.html>). *D Programming Language 1.0*. Digital Mars. . Retrieved 1 December 2011.
- [2] "Changelog" (<http://dlang.org/changelog.html>). *D Programming Language 2.0*. Digital Mars. . Retrieved 3 August 2012.
- [3] "Changelog" (<http://www.digitalmars.com/d/1.0/changelog.html>). *D Programming Language 1.0*. Digital Mars. . Retrieved 3 August 2012.
- [4] "dmd 1.075 and 2.060 release" ([http://forum.dlang.org/post/jvejr8\\$2s31\\$1@digitalmars.com](http://forum.dlang.org/post/jvejr8$2s31$1@digitalmars.com)). . Retrieved 3 August 2012.
- [5] "readme.txt" (<https://github.com/D-Programming-Language/dmd/blob/master/src/readme.txt>). *DMD source code*. GitHub. . Retrieved March 5, 2012.
- [6] FAQ of digitalmars (<http://dlang.org/faq.html>)
- [7] Andrei Alexandrescu (August 2, 2010). *Three Cool Things About D* (<http://www.youtube.com/watch?v=RIVpPstLPEc>). .
- [8] "std.gc" (http://www.digitalmars.com/d/1.0/phobos/std_gc.html). *D Programming Language 1.0*. Digital Mars. . Retrieved 6 July 2010.
- [9] "Memory Management" (<http://dlang.org/memory.html>). *D Programming Language 2.0*. Digital Mars. . Retrieved February 17, 2012.
- [10] "D Change Log" (<http://www.digitalmars.com/d/1.0/changelog2.html>). *D Programming Language 1.0*. Digital Mars. . Retrieved 11 January 2012.
- [11] "Intro" (<http://www.digitalmars.com/d/1.0/>). *D Programming Language 1.0*. Digital Mars. . Retrieved 1 December 2011.
- [12] "Announcing a new library" ([http://forum.dlang.org/post/en9ou4\\$23hr\\$1@digitaldaemon.com](http://forum.dlang.org/post/en9ou4$23hr$1@digitaldaemon.com)). . Retrieved 15 February 2012.
- [13] "Wiki4D - Standard Lib" (<http://www.prowiki.org/wiki4d/wiki.cgi?StandardLib>). . Retrieved 6 July 2010.
- [14] "Tango for D2: All user modules ported" ([http://forum.dlang.org/post/jgagrl\\$1ta5\\$1@digitalmars.com](http://forum.dlang.org/post/jgagrl$1ta5$1@digitalmars.com)). . Retrieved 16 February 2012.
- [15] Walter Bright. "Re: GitHub or dsources?" ([http://forum.dlang.org/post/iv524m\\$98r\\$1@digitalmars.com](http://forum.dlang.org/post/iv524m$98r$1@digitalmars.com)). . Retrieved 15 February 2012.
- [16] "gdc project homepage" (<http://gdcproject.org/>). . Retrieved 14 October 2012.
- [17] "LLVM D compiler project on DSource" (<http://dsource.org/projects/ldc>). . Retrieved 3 July 2010.
- [18] (<http://www.dsoucre.org/projects/ldc/wiki/BuildInstructionsPhobosDruntimeTrunk>)
- [19] "D .NET project on CodePlex" (<http://dnet.codeplex.com/>). . Retrieved 3 July 2010.
- [20] Jonathan Allen (15 May 2009). "Source for the D.NET Compiler is Now Available" (<http://www.infoq.com/news/2009/05/D-Source>). InfoQ. . Retrieved 6 July 2010.
- [21] Zeus (<http://www.prowiki.org/wiki4d/wiki.cgi?EditorSupport/ZeusForWindows>)
- [22] "Wiki4D - Editor Support" (<http://www.prowiki.org/wiki4d/wiki.cgi?EditorSupport>). . Retrieved 3 July 2010.
- [23] DDT (<http://code.google.com/a/eclipselabs.org/p/ddt/>)
- [24] Descent (<http://dsoucre.org/projects/descent>)
- [25] VisualD (<http://www.dsoucre.org/projects/visuald>)
- [26] D for Xcode (<http://michelf.com/projects/d-for-xcode/>)
- [27] Mono-D (<http://mono-d.sourceforge.net/>)
- [28] Poseidon (<http://dsoucre.org/projects/poseidon>)
- [29] D-IDE (<http://d-ide.sourceforge.net/>)
- [30] Entice Designer (<http://www.dprogramming.com/entice.php>)
- [31] <http://ddbg.mainia.de/>
- [32] <http://dsoucre.org/projects/cv2pdb>
- [33] <http://zerobugs.codeplex.com/>

Further reading

- Alexandrescu, Andrei (January 4, 2010). *The D Programming Language* (1 ed.). Addison-Wesley Professional. ISBN 978-0-321-63536-5.
- Alexandrescu, Andrei (June 15, 2009). "The Case for D" (<http://www.ddj.com/hpc-high-performance-computing/217801225>). Dr. Dobb's Journal.
- Çehreli, Ali (February 1, 2012). "Programming in D" (<http://ddili.org/ders/d.en/index.html>). (distributed under CC-BY-NC-SA license). It's basic level introduction.

External links

- Official website (<http://dlang.org>)
- Digital Mars (<http://www.digitalmars.com>)
- D on GitHub (<https://github.com/D-Programming-Language>)

Go (programming language)

Go

Paradigm(s)	compiled, concurrent, imperative, structured
Appeared in	2009
Designed by	Robert Griesemer Rob Pike Ken Thompson
Developer	Google Inc.
Stable release	version 1.0.3 ^[1] (24 September 2012)
Typing discipline	strong, static
Major implementations	gc (8g, 6g, 5g), gccgo
Influenced by	C, Limbo, Modula, Newspeak, Oberon, Pascal, ^[2] Python
OS	Linux, Mac OS X, FreeBSD, OpenBSD, MS Windows, Plan 9 ^[3]
License	BSD-style ^[4] + Patent grant ^[5]
Usual filename extensions	.go
Website	golang.org ^[6]

Go is a compiled, garbage-collected, concurrent programming language developed by Google Inc.^[7]

The initial design of Go was started in September 2007 by Robert Griesemer, Rob Pike, and Ken Thompson.^[2] Go was officially announced in November 2009. In May 2010, Rob Pike publicly stated that Go was being used "for real stuff" at Google.^[8] Go's "gc" compiler targets the Linux, Mac OS X, FreeBSD, OpenBSD, Plan 9, and Microsoft Windows operating systems and the i386, amd64, and ARM processor architectures.^[9]

Goals

Go aims to provide the efficiency of a statically typed compiled language with the ease of programming of a dynamic language.^[10] Other goals include:

- Safety: Type-safe and memory-safe.
- Good support for concurrency and communication.
- Efficient, latency-free garbage collection.
- High-speed compilation.

Description

The syntax of Go is broadly similar to that of C: blocks of code are surrounded with curly braces; common control flow structures include `for`, `switch`, and `if`. Unlike C, line-ending semicolons are optional, variable declarations are written differently and are usually optional, type conversions must be made explicitly, and new `go` and `select` control keywords have been introduced to support concurrent programming. New built-in types include maps, UTF-8 strings (without proper Unicode strings abstraction^[11]), array slices, and channels for inter-thread communication.

Go is designed for exceptionally fast compiling times, even on modest hardware.^[12] The language requires garbage collection. Certain concurrency-related structural conventions of Go (channels and alternative channel inputs) are borrowed from Tony Hoare's CSP. Unlike previous concurrent programming languages such as occam or Limbo, Go does not provide any built-in notion of safe or verifiable concurrency.^[13]

Of features found in C++ or Java, Go does not include type inheritance, generic programming, assertions, method overloading, or pointer arithmetic.^[2] Of these, the Go authors express an openness to generic programming, explicitly argue against assertions and pointer arithmetic, while defending the choice to omit type inheritance as giving a more useful language, encouraging heavy use of interfaces instead.^[2] Initially, the language did not include exception handling, but in March 2010 a mechanism known as `panic/recover` was implemented to handle exceptional errors while avoiding some of the problems the Go authors find with exceptions.^{[14][15]}

Type system

Go allows a programmer to write functions that can operate on inputs of arbitrary type, provided that the type implements the functions defined by a given interface.

Unlike Java, the interfaces a type supports do not need to be specified at the point at which the type is defined, and Go interfaces do not participate in a type hierarchy. A Go interface is best described as a set of methods, each identified by a name and signature. A type is considered to implement an interface if all the required methods have been defined for that type. An interface can be declared to "embed" other interfaces, meaning the declared interface includes the methods defined in the other interfaces.^[13]

Unlike Java, the in-memory representation of an object does not contain a pointer to a virtual method table. Instead a value of interface type is implemented as a pair of a pointer to the object, and a pointer to a dictionary containing implementations of the interface methods for that type.

Consider the following example:

```
type Sequence []int

func (s Sequence) Len() int {
    return len(s)
}

type HasLength interface {
    Len() int
}

func Foo (o HasLength) {
    ...
}
```

These four definitions could have been placed in separate files, in different parts of the program. Notably, the programmer who defined the `Sequence` type did not need to declare that the type implemented `HasLength`, and the person who implemented the `Len` method for `Sequence` did not need to specify that this method was part of `HasLength`.

Name visibility

Visibility of structures, structure fields, variables, constants, methods, top-level types and functions outside their defining package is defined implicitly according to the capitalization of their identifier.^[16]

Concurrency

Go provides *goroutines*, small lightweight threads; the name alludes to coroutines. Goroutines are created with the `go` statement from anonymous or named functions.

Goroutines are executed in parallel with other goroutines, including their caller. They do not necessarily run in separate threads, but a group of goroutines are multiplexed onto multiple threads — execution control is moved between them by blocking them when sending or receiving messages over channels.

Implementations

There are currently two Go compilers:

- 6g/8g/5g (the compilers for AMD64, x86, and ARM respectively) with their supporting tools (collectively known as "gc") based on Ken's previous work on Plan 9's C toolchain.
- gccgo, a GCC frontend written in C++,^[17] and now officially supported as of version 4.6, albeit not part of the standard binary for gcc.^[18]

Both compilers work on Unix-like systems, and a port to Microsoft Windows of the gc compiler and runtime have been integrated in the main distribution. Most of the standard libraries also work on Windows.

There is also an unmaintained "tiny" runtime environment that allows Go programs to run on bare hardware.^[19]

Examples

Hello world

The following is a Hello world program in Go:

```
package main

import "fmt"

func main() {
    fmt.Println("Hello, World")
}
```

Go's automatic semicolon insertion feature requires that opening braces not be placed on their own lines, and this is thus the preferred brace style; the examples shown comply with this style.^[20]

Echo

Example illustrating how to write a program like the Unix echo command in Go:^[21]

```
package main

import (
    "os"
    "flag" // Command line option parser.
)
```

```

var omitNewline = flag.Bool("n", false, "don't print final newline")

const (
    Space = " "
    Newline = "\n"
)

func main() {
    flag.Parse() // Scans the arg list and sets up flags.
    var s string
    for i := 0; i < flag.NArg(); i++ {
        if i > 0 {
            s += Space
        }
        s += flag.Arg(i)
    }
    if !*omitNewline {
        s += Newline
    }
    os.Stdout.WriteString(s)
}

```

Reception

Go's initial release led to much discussion.

Michele Simionato wrote in an article for artima.com:^[22]

Here I just wanted to point out the design choices about interfaces and inheritance. Such ideas are not new and it is a shame that no popular language has followed such particular route in the design space. I hope Go will become popular; if not, I hope such ideas will finally enter in a popular language, we are already 10 or 20 years too late :-(

Dave Astels at Engine Yard wrote:^[23]

Go is extremely easy to dive into. There are a minimal number of fundamental language concepts and the syntax is clean and designed to be clear and unambiguous. Go is still experimental and still a little rough around the edges.

Ars Technica interviewed Rob Pike, one of the authors of Go, and asked why a new language was needed. He replied that:^[24]

It wasn't enough to just add features to existing programming languages, because sometimes you can get more in the long run by taking things away. They wanted to start from scratch and rethink everything. ... [But they did not want] to deviate too much from what developers already knew because they wanted to avoid alienating Go's target audience.

Go was in 15th place on the TIOBE Programming Community Index of programming language popularity in its first year, 2009, surpassing established languages like Pascal. As of March 2012, it ranked 66th in the index.^[25]

Bruce Eckel stated:^[26]

The complexity of C++ (even more complexity has been added in the new C++), and the resulting impact on productivity, is no longer justified. All the hoops that the C++ programmer had to jump through in order to use

a C-compatible language make no sense anymore -- they're just a waste of time and effort. Now, Go makes much more sense for the class of problems that C++ was originally intended to solve.

Naming dispute

On the day of the general release of the language, Francis McCabe, developer of the Go! programming language (note the exclamation point), requested a name change of Google's language to prevent confusion with his language.^[27] The issue was closed by a Google developer on 12 October 2010 with the custom status "Unfortunate", with a comment that "there are many computing products and services named Go. In the 11 months since our release, there has been minimal confusion of the two languages."^[28]

References

- [1] Gerrand, Andrew (24 September 2012). "go1.0.3 released" (<https://groups.google.com/forum/#topic/golang-announce/co3SvXbGrNk>). *golang-announce mailing list*. Google. . Retrieved 05 October 2012.
- [2] "Language Design FAQ" (http://golang.org/doc/go_faq.html). *golang.org*. 16 January 2010. . Retrieved 27 February 2010.
- [3] "Go Porting Efforts" (<http://go-lang.cat-v.org/os-ports>). *Go Language Resources*. cat-v. 12 January 2010. . Retrieved 18 January 2010.
- [4] "Text file LICENSE" (<http://golang.org/LICENSE>). *The Go Programming Language*. Google. . Retrieved 05 October 2012.
- [5] "Additional IP Rights Grant" (<http://code.google.com/p/go/source/browse/PATENTS>). *The Go Programming Language*. Google. . Retrieved 05 October 2012.
- [6] <http://golang.org>
- [7] Kincaid, Jason (10 November 2009). "Google's Go: A New Programming Language That's Python Meets C++" (<http://www.techcrunch.com/2009/11/10/google-go-language/>). *TechCrunch*. . Retrieved 18 January 2010.
- [8] Metz, Cade (20 May 2010). "Google programming Frankenstein is a Go" (http://www.theregister.co.uk/2010/05/20/go_in_production_at_google/). *The Register*.
- [9] "Installing Go" (http://golang.org/doc/install.html#tmp_33). *golang.org*. The Go Authors. 11 June 2010. . Retrieved 11 June 2010.
- [10] Pike, Rob. "The Go Programming Language" (<http://www.youtube.com/watch?v=rKnDgT73v8s&feature=related>). YouTube. . Retrieved 1 Jul 2011.
- [11] The importance of language-level abstract Unicode strings « Unspecified Behaviour (<http://unspecified.wordpress.com/2012/04/19/the-importance-of-language-level-abstract-unicode-strings/>)
- [12] Rob Pike (10 November 2009) (flv). *The Go Programming Language* (<http://www.youtube.com/watch?v=rKnDgT73v8s#t=8m53>) (Tech talk). Google. Event occurs at 8:53. .
- [13] "The Go Memory Model" (http://golang.org/doc/go_mem.html). Google. . Retrieved 5 January 2011.
- [14] Release notes, 30 March 2010 (<http://golang.org/doc/devel/weekly.html#2010-03-30>)
- [15] "Proposal for an exception-like mechanism" (http://groups.google.com/group/golang-nuts/browse_thread/thread/1ce5cd050bb973e4). *golang-nuts*. 25 March 2010. . Retrieved 25 March 2010.
- [16] "A Tutorial for the Go Programming Language" (http://golang.org/doc/go_tutorial.html). *The Go Programming Language*. Google. . Retrieved 10 March 2010. "In Go the rule about visibility of information is simple: if a name (of a top-level type, function, method, constant or variable, or of a structure field or method) is capitalized, users of the package may see it. Otherwise, the name and hence the thing being named is visible only inside the package in which it is declared."
- [17] "FAQ: Implementation" (http://golang.org/doc/go_faq.html#Implementation). *golang.org*. 16 January 2010. . Retrieved 18 January 2010.
- [18] "Installing GCC: Configuration" (<http://gcc.gnu.org/install/configure.html>). . Retrieved 3 December 2011. "Ada, Go and Objective-C++ are not default languages"
- [19] Gerrand, Andrew (1 February 2011). "release.2011-02-01" (http://groups.google.com/group/golang-nuts/browse_thread/thread/b877e34723b543a7). *golang-nuts*. Google. . Retrieved 5 February 2011.
- [20] "A Tutorial for the Go Programming Language" (http://golang.org/doc/go_tutorial.html). *The Go Programming Language*. Google. . Retrieved 10 March 2010. "The one surprise is that it's important to put the opening brace of a construct such as an if statement on the same line as the if; however, if you don't, there are situations that may not compile or may give the wrong result. The language forces the brace style to some extent."
- [21] "A Tutorial for the Go Programming Language" (http://golang.org/doc/go_tutorial.html). *golang.org*. 16 January 2010. . Retrieved 18 January 2010.
- [22] Simionato, Michele (15 November 2009). "Interfaces vs Inheritance (or, watch out for Go!)" (<http://www.artima.com/weblogs/viewpost.jsp?thread=274019>). artima. . Retrieved 15 November 2009.
- [23] Astels, Dave (9 November 2009). "Ready, Set, Go!" (<http://www.engineyard.com/blog/2009/ready-set-go/>). engineyard. . Retrieved 9 November 2009.

- [24] Paul, Ryan (10 November 2009). "Go: new open source programming language from Google" (<http://arstechnica.com/open-source/news/2009/11/go-new-open-source-programming-language-from-google.ars>). Ars Technica. . Retrieved 13 November 2009.
- [25] "TIOBE Programming Community Index for March 2012" (<http://es.scribd.com/doc/89569304/TIOBE-Programming-Community-Index-for-March-2012>). TIOBE Software. March 2012. . Retrieved 28 April 2012.
- [26] Bruce Eckel (27). "Calling Go from Python via JSON-RPC" (<http://www.artima.com/weblogs/viewpost.jsp?thread=333589>). . Retrieved 29 August 2011.
- [27] Claburn, Thomas (11 November 2009). "Google 'Go' Name Brings Accusations Of Evil" (http://www.informationweek.com/news/software/web_services/showArticle.jhtml?articleID=221601351). InformationWeek. . Retrieved 18 January 2010.
- [28] "Issue 9 - go - I have already used the name for *MY* programming language" (<http://code.google.com/p/go/issues/detail?id=9>). *Google Code*. Google Inc.. . Retrieved 12 October 2010.

Further reading

- Chisnall, David (9 May 2012). "Common Go Patterns" (<http://www.informit.com/articles/article.aspx?p=1760496>). *The Go Programming Language Phrasebook*. Addison-Wesley Professional. ISBN 0-321-81714-1.
- Summerfield, Mark (5 May 2012). *Programming in Go: Creating Applications for the 21st Century* (<http://www.informit.com/store/product.aspx?isbn=0321774639>). Addison-Wesley Professional. ISBN 0-321-77463-9.

External links

- Official website (<http://golang.org>)
- Pike, Rob (28 April 2010). "Another Go at Language Design" (<http://www.stanford.edu/class/ee380/Abstracts/100428.html>). *Stanford EE Computer Systems Colloquium*. Stanford University. (video (<http://ee380.stanford.edu/cgi-bin/videologger.php?target=100428-ee380-300.asx>) — A university lecture
- Wynn Netherland & Adam Stacoviak (27 November 2009). "Episode 0.0.3 - Google's Go Programming Language" (<http://thechangelog.com/post/259401776/episode-0-0-3-googles-go-programming-language>). *The Changelog* (Podcast). — Interview with Rob Pike, Tech Lead for the Google Go team
- Go Programming Language Resources (<http://go-lang.cat-v.org/>) (unofficial)
- irc://chat.freenode.net/#go-nuts – the IRC channel #go-nuts on freenode
- Steve Dalton (22 January 2011). "Episode 20 (Interview with Andrew Gerrand about Go Programming Language)" (<http://www.codingbynumbers.com/2011/01/coding-by-numbers-episode-20-interview.html>). *Coding By Numbers* (Podcast).
- Schuster, Werner (25 February 2011). "Rob Pike on Google Go: Concurrency, Type System, Memory Management and GC" (<http://www.infoq.com/interviews/pike-google-go>). *InfoQ*. GOTO Conference: C4Media Inc..

Article Sources and Contributors

Programming language *Source:* <http://en.wikipedia.org/w/index.php?oldid=515147282> *Contributors:* -Barry-, 10.7, 151.203.224.xxx, 16@r, 198.97.55.xxx, 199.196.144.xxx, 1exec1, 203.81.xxx, 212.188.19.xxx, 2988, 96.186, A520, AJim, Abednigo, Abeliaovsky, Abram.carolan, Acacix, Acaciz, AccurateOne, Addicted2Sanity, Ahoerstemeier, Ahyl, Akadruid, Alansohn, Alex, AlexPlank, Alhoori, Alksub, Allan McInnes, Alliswellthen, Altenmann, Amire80, Ancheta Wis, Andonic, Andre Engels, Andres, AndyMurphy, Angel, Angusmclellan, Antonielly, Ap, Apwestern, ArmadilloFromHell, AstroNomer, Autocratique, Avono, B4hand, Behnam, Beland, Ben Ben, Standeven, Benjaminct, Bevo, Bh3u4m, BigSmoke, Bill122, BioPupil, BirgitteSB, Blaisorblade, Blanchardb, Bobblewik, Bobo192, Bonaovox, Booyabazooka, Borislav, Brandon, Brentdax, Brianjd, Brick Thrower, Brion VIBBER, Bubba73, Burkedavis, CSProfBill, Caltech, Can't sleep, clown will eat me, CanisRufus, Capricorn42, Captain-n00dle, CarlHewitt, Carmichael, Catgut, Centrx, Charlesriver, Charlie Huggard, Chillum, Chinabuffalo, Chun-hian, Cireshoe, Ckatz, Closedmouth, Cmichael, Cobaltbluetony, ColdFeet, Conor123777, Conversion script, Cp15, Cybercobra, DBigXray, DMacks, DVD R W, Damian.rouson, Damieng, Dan128, Danakil, Danim, Dave Bell, David, Monniaux, DavidHOzAu, Davidstr, Davidpdx, Dcoeteet, DeadEyeArrow, DenisMoskowitz, DennisDaniels, DerHexer, Derek Ross, Derek.farn, Dianmaa, Diego Moya, Dolfrog, Dominator09, Don't Copy That Floppy, Donhalcon, Doradus, DouglasGreen, Dreftymac, Dtaylor1984, Duke Ganote, Dysprosia, EJF, ESkoog, EagleOne, Edward301, Eivind F Øyangen, ElAmericano, Elembis, EncMstr, EngineerScotty, Epb123, Esap, Evercat, Everyking, Ewlyahooocom, Ezrakility, Fantom, Faradayplank, Fayt82, Fielday-sunday, Finlay McLWalter, Fl, Foobah, Four Dog Night, Fplay, Fraggle81, Fredrik, Friedo, Fubar Obfusco, Funandtrvl, FvdP, Gaius Cornelius, Galoubet, Gazpacho, Gbruin, Georg Peter, Giftlife, Giorgios, Giotto, Goodgerster, Gploc, Green caterpillar, GregAsche, Grin, Gurch, Gutzke, Hadil, Hairy Dude, Hammer1980, Hans Adler, HarisM, Harmil, Hayabusa future, Headbomh, HeikoEvermann, HenryLi, Hfastedge, HopeChrist, Hoziron, Hut 8.5, Hyad, INKbusse, IanOsgood, Icey, Ideogram, Ilario, Imran, Indon, Infinoid, Iridescent, Iwantitalllllll, Ifxd64, J.delanoy, JMSwtlk, JPINFV, JaK81600, JanSuchy, Jarble, JasonSayers, Jaxad0127, Jaxl, Jeffrey Mall, Jeltz, Jeronimo, Jerryobject, Jgyu, Jim1138, Jitse Niesen, Jj137, Johana Wolfgang, John lindgren, John254, JohnLai, JohnWittle, Jonik, Jorend, Jossi, Joyous!, Jpbown, Jpk, Jschnur, JulesH, Juliancolton, Jusjil, Jwissick, K.lee, K12308025, KHaskell, KSmrq, KTC, Karingo, Karthikndr, Katielh5584, Kbdbank71, Kbh3rd, Kedearian, Ketiltrottr, Khalid Mahmood, Kickstart70, Kiminatheguardian, Kimse, Kinema, Klasbricks, KnowledgeOfSelf, Knfy, Kooginup, Koyaanis Qatsi, Kraken, Krauss, Krawi, Kris Schnee, Krischik, Kuciwalker, Kungfuadum, Kwertii, KymFarnik, L.Gottschalk, L33tmision, LC, Lagalag, Leibniz, Liao, Lightmouse, Ligulem, LindsayH, LinguistAtLarge, Logarkh, LordCo Centre, Lradrama, Lucian1900, Lulu of the Lotus-Eaters, Luna Santin, Lupo, MER-C, MK8, Mac c, Macaldo, Macrakis, Magnus Manske, Mahanga, Majilis, Malcolm Farmer, Malleus Faturorum, Mangojuice, Manpreet, Marcos, Mark Renier, MarsRover, MartinHarper, MartyMcGowan, Marudubshini, Materialscientist, Matthew Woodcraft, Mattisse, Max, Maxis ftw, McSly, Mccready, Mean as custard, MearsMan, MegaHasher, Mellum, Mendaliv, Merbabu, Merphant, Michael Hardy, Michael Zimmermann, Midinatasurazz, Mike Rosoff, Mild Bill Hiccup, Minesweeper, MisterCharlie, Miym, Mkdw, Monz, Mpils, Mrdeep, Mrjeff, Ms2ger, Mschel, Muro de Aguas, Murray Langton, Mwaiberg, Mn, Mithrandir, NSiln, Naderi 8189, Nameneko, Nanshu, Napi, Natalie Erin, Natkeeran, NawlinWiki, Nbrosthers, Necklace, NewEnglandYankee, NewbieDoo, Nick125, Nikai, Nima1024, Ningauble, Nixdorf, Noctibus, Noformation, Noisy, Noldoaran, Noosental, NotQuiteEXPComplete, Nottsadol, Novasource, Ntalamai, Nuggetboy, Nutsnbolts222, Oblivious, Ohms law, Ohnoitsjamie, Oldadamm, Oleg Alexandrov, Omphaloscope, Orderud, OrgasGirl, Orphan Wiki, Papercutbiology, Paul August, PaulFord, Pcap, Peter, PeterPerdjones, PhilSandifer, PhilKnight, Philg88, Photonique, Phyzome, Pieguy82, Piet Delport, PlayStation 69, Poor Yorick, PoorYorick, Positron, Prolog, PtkPumpie, Pv1, Quagmire, Quiddity, Quota, Quuixplusone, Qwyrixian, RainerBlome, Raise exception, Ranafon, RayAYang, RedWolf, Reddi, Reelrt, Reinis, RenamedUser2, Revived, RexNL, Rich Farmbrough, Rjstott, Rjwilmsi, Rlee0001, Robbe, Robert A West, Robert Skyhawk, Robo Cop, Roland2, Romanm, Ronjhones, Roux, Royboycrashfan, Rrburke, Rursus, Rushyo, Russell Joseph McCann, Ruud Koot, S.Örvarr.S, Saccade, Sam Korn, Science History, Seaphoto, SeeAnd, Sekelsenmat, Sgbirch, Shadowjams, Shane A Bender, Shanes, ShellSkewed, SimonP, Simplyanil, Sjakkalle, Skytreder, Slaad, Slakr, Slashem, SmartBee, Snickel11, Sonicology, SparsityProblem, Spec112, Speed Air Man, SpeedyGonsales, Speuler, SpuriousQ, Stephen B Steatner, Stephenb, SubSeven, Suffusion of Yellow, Suruena, Swatiri, Swirsky, Switcherat, Systemetsky, TakuyaMurata, Tarret, Taxman, Techman224, Tedickey, Template namespace initialisation script, Teval, Tewy, Tgeairn, Tgr, The Thing That Should Not Be, TheTechFan, Thniels, Thomasuniko, Thv, Tiddly Tom, Tide rolls, Tim Starling, Timhowardiley, Tizio, Tobias Bergemann, TomTdm, Tomatensaft, Tony1, TonyClarke, Torc2, Toussaint, Trusilver, Tuukkah, Tysto, Ubiq, Ulric1313, Ultra two, Undefrence, Useight, Usman&muzammal, Vadmium, Vahid83, Vaibhankwanal, Vald, VampWillow, VictorAnyakin, Victorriggas, Vivin, Vkhaitan, Vriullop, VSION, WAS 4.250, Waterfles, Wavelength, Wiki alf, Wiki13, WikiTome, Wikiboh, Wikiboh, Wikisedia, Wimt, Windharp, Wlievens, Wmahan, Woohookitty, Ww, Xaosflux, Xavier Combelle, Yana209, Yath, Yk Yk Yk, Yoric, Zaheen, Zarniwoot, Zawersh, ZeWrestler, Zero1328, Zoicon5, Zondor,²¹², ², ², 927 anonymous edits

Programming paradigm *Source:* <http://en.wikipedia.org/w/index.php?oldid=520538093> *Contributors:* "alyosha", :Ajvol:, AGK, AThing, Aarsanjani, AbramDemski, Adamd1008, Aeternus, Agnustus, AleksanderVatov, Alfio, Alismayilov, Amoss, Ancheta Wis, Angr, Antonielly, Ap, Aphoton, Arabic Pilot, ArneBab, Arny, Associat0, Badly Bradley, BenFrantzDale, Bendlund, Benwing, Bkell, BlueAmethyst, Bonus Onus, BrainCricket, BrianDeacon, Britannica, Bunnyhop11, Burschik, CRGreathouse, Carbo1200, CarlHewitt, Carleas, Chalst, Chiswick Chap, Christian.heller, Clubmax, Cogiat, Comper, Cowlfsheep, Cybercobra, Danakil, David Eppstein, Decltype, Demitsu, Demonkory, Dianoetic, Diego Moya, DragonRdr, Ejhewy, Elkman, Elockid, Eng.ahmed, Faradayplank, FatalError, Fentlehan, FlyHigh, Fredrik, Fubar Obfusco, Fyrael, Geregen2, GoShow, Golden herring, Gragrara, Gruu, Gwizard, Hede2000, Hmains, HoodedMan, Isaak Sanolnacov, JDubman, Jdh30, Jerrolet, Jerryobject, Jlochoap, JohnJSal, Jonnyapple, JulesH, Junegold, K.lee, Kdakin, Kendrick Hang, Kenny Moens, Knuckles, Kooginup, LedgendGamer, LinguistAtLarge, Loadmaster, Luís Felipe Braga, Madmardigan53, Mark Renier, MarkKallos31, MartinHarper, MartinSpamer, Maxim.mazin, Mdd, Mean as custard, Metalpasma, Michael Hardy, Msreeharsha, Murray Langton, Musiphil, Nath1991, Natkeeran, Nbarth, Neazp, Nerak99, Ninly, Nixdorf, Njk, Norandav, Oicumayberight, Oleg Alexandrov, Omicronperse8, Only2sea, Orang Hutan, Orderud, Ospalh, ParkerJones2007, Pathoschild, Pcap, Pengo, Perlyn006, Pete Wall, Peter Van Roy, PhnToM89, Pheeror, Pinar, Pluke, Ptb, Quuixplusone, Randomious, Retired username, Robofish, Robe, RoyArne, Rumping, Rursus, Ruud Koot, S.K., Saquigle, Seitzedave, Semifinalist, Sergey Dmitriev, Shinji14, Sibian, SkyCaptain, Steveozone, Sunray, Suruena, TakuyaMurata, Tbtiect, Those words, Tonyfaul, Torc2, Torqueing, Totake423, Toussaint, TravisMusnon1993, Tuukkah, Uucp, Wavehunter, Wham Bam Rock II, Wlievens, WriterHound, Yhever, Zeus-chu, Zickzack, Zorabi, Zowie, ^{کشف فغل}, 219 anonymous edits

Procedural programming *Source:* <http://en.wikipedia.org/w/index.php?oldid=514946986> *Contributors:* 16@r, A Keshavarz, AJR, Aarghdaark, Ajh16, Alfredo ougaowen, Ancheta Wis, Andrew Eisenberg, AndrewHowse, AstroPig7, AxelBoldt, Baseballduide, Beakerboy, Beland, Beliavsky, Benandersqueaks, Bevo, Blonkm, Bryan Derksen, Burchard, Burschik, CALR, CONFIQ, CambridgeBayWeather, CapitalR, CapitalSasha, Chbarts, Cheshins, ClickRick, Colonies Chris, Conversion script, CrazyMYKL, Cybercobra, DARTH SIDIOUS 2, DKEwards, Danakil, Davidguitierrezvalarez, Davou, Deewiant, Denispix, Dex1337, Dylannmd, ErkinBatu, EugeneZelenko, FatalError, Felixdakat, Gazpacho, Gilliam, HKT, Hmrox, Ian Pitchford, JAF1970, JDowning, JLATondre, Jalesh, Jamelan, Jerryobject, Jethro B, Jobers, Joe Sewell, Jons63, Jorgenev, Jtaylorov, Kavadi carrier, Keith D, Kowey, Kusunose, Kylehuang, LOL, LeaveSleaves, LiHelpa, LinguistAtLarge, Logperson, Loudsox, MaNeMeBasat, Marco Krohn, Mark91, Mastazi, Max42, Md bluelily, Meanskeeps, Mmortal03, Mnsc, Modulatum, Mrtrumbe, Msikma, Ndenison, Nixdorf, Only2sea, Or-whatever, OrangUtan, Orderud, PJTrail, Ph.eyes, Pharaoh of the Wizards, PhilKnight, Pnm, PradeepArya1109, Rabooft, Rade Kutil, Redaktor, Renku, Rgamble, Rhopkins8, Rich Farmbrough, Rhee0001, Ruud Koot, Ryuukuro, Sean Whiton, Shadowcode, Sibidiba, Simon80, SimonTrew, SkyWalker, Sykopomp, TakuyaMurata, Taw, The Anome, The Rambling Man, The Wordsmith, The sock that should not be, Thivier, Thsgm, Thumperward, Torc2, Torqueing, Totake423, Toussaint, TravisMusnon1993, Tuukkah, Uucp, VKokielov, Valfontis, WalterGR, Wfox, Windharp, Work permit, Yacoubean, Yuanchosaan, ZMaen, ZeroOne, ^{امحمد مصطفی السيد}, 283 anonymous edits

Structured programming *Source:* <http://en.wikipedia.org/w/index.php?oldid=510757962> *Contributors:* 203.37.81.xxx, AJim, Aaron Nitro Danielson, Aeosynth, AlanUS, Aledubr, Angrytoast, AxelBoldt, Bloodshedder, Boing! said Zebedee, BraneJ, Brichard37, Brusselshrek, Bryan Derksen, Burschik, Cactus26, Centrx, Conversion script, CppClimber, Crazzytales, Cspfdmone, Damian Yerrick, Danakil, Derek Ross, Doug, Dphsmith, Egerm, Elementologist, EngineerScotty, Enharmonix, Fram, Furykef, Gazpacho, Ghepeu, Gillwoman, Graemel, Hans Dunkelberg, Harris7, Iccaldwell, InverseHypercube, IPalonus, Jarble, Jeronimo, Jim328, JonathanCross, Jervine, Kdakin, Kenny sh, Kim Bruning, Knutux, Krischik, Leandro, LinguistAtLarge, LittleDan, Loadmaster, Lph, Magnhus, Mdd, Michael Hardy, Modster, Moe Aboulkheir, Mr MaRo, Mulad, Murray Langton, Nanshu, Netoholic, Nikai, NipplesMeCool, Nwerneck, Octavia.china, Oddity-, Orion 8, PJTrail, Panairjde, Paoloss, PatheticCopyEditor, Pcap, Porges, Psb777, R.e.s., RTC, RainbowOfLight, Reflex Reaction, Rene Mas, Renku, Rrburke, SMC, SantiagoGala, SkyWalker, Smyth, Snori, Spoon!, Stephemb, Stevage, Stormie, Subversive.sound, Taw, Tedickey, Tjdw, Tobias Bergemann, Tomo, Toussaint, Trivialist, Trusilver, Ugncreative Username, Unfree, Vipinhar, Wavelength, WikHead, Wikid77, Xen 1986, YordanGeorgiev, Ziusudra, Zron, ^{احمد مصطفى السيد}, 141 anonymous edits

Fortran *Source:* <http://en.wikipedia.org/w/index.php?oldid=519798475> *Contributors:* 16@r, 1exec1, Aaron Schulz, Abeliavsky, AdjustShift, Agatelle, Agricola44, Aivosto, Alex Blokha, Amos Wolfe, Anarchivist, Ancantis, Andre Engels, Andrej, Arch dude, ArnoldReinhold, Atlant, Audriusa, AxelBoldt, Bachcell, Badger Drink, Bashen, Bchabal2, Bduke, Beliavsky, BenFrantzDale, Bender235, Benjaminevans82, Betacommand, Bethlehem4, Between My Ken, Bevo, Bitesmart, Blaxthos, Bobdc, Brad Hughes, Brion VIBBER, Broh., Bryan, Buba73, Bunnyhop11, CMW275, CRGreathouse, CYD, Captain Fortran, Casey56, Changcho, ChaoaCon, Chbars, ChongDae, Chris, ChrisGualtieri, Closeapple, Conversion script, Corti, Corvus cornix, Cromis, Crumley, Csabo, Cybercobra, D. F. Schmidt, DBRane, DMacks, DNewhall, DRady, Damian Yerrick, Damian.rouson, Dan100, Danakil, Danim, Dav4is, David Biddulph, David.Monniaux, Dcoeteet, Dead3y3, Dejvid, Dekart, Dennette, DerHexer, Derek Ross, Derek.farn, DesertAngel, Dimsar, DoctorWho42, Dpbsmith, DragonHawk, Drilnoth, Drj, Duja, Dune, Dycedarg, Dysprosia, Eagleal, Eloash, EneMstr, Eprb123, Ericamic, Erschnet, Eubulides, Eustress, Evolauxa, Ewlyahooocom, Falcon8765, Fireutzer, Flex, Flowerheaven, Frecklefoot, Funandtrvl, Furykef, Fa, Gain Cornelius, Gantlord, Gareth Owen, Gbchuna, Geoff97, Georg Peter, GeorgeMoney, Ghettoblaster, Gjd001, Glenn, Graham87, Greg L, Greg Lindahl, Gringer, Gcschoryu, Gudeldar, H2g2bob, Hairy Dude, Herbee, Howardjp, Howcheng, Hydrargyrum, I already forgot, I dream of horses, II MusLiM HyBRID II, Infologue, Intgr, Inwind, Iridescent, Ironmagma, IsaacGS, Isis, Iulian.serbanoiu, Ifxd64, JW1805, Jacobolus, Jamesgibbon, Jarble, Jerryobject, Jim1138, JonEahlquist, Jossi, Jpfagerback, Jrockley, Julesd, Jvhertum, Jwmwalrus, Jwoodger, Jww19, K-cat, Kajasaduhakarabu, Karl-Henner, Kbh3rd, Keflavich, Kenyon, Kglavin, KieferSkunk, Kleb, Knebel, KymFarnik, LOL, LOTRules, LanceBarber, Larsivi, Leandro, Lerdwsu, Liao, Ligulem, Little Mountain 5, Ljn917, Lotje, Lousyd, Luminifer, Magnhus, Malaki1874, MarcusMaximus, Mark Foskey, MarkSweep, MathChem271828, Mathmo, MattGiua, Matthewdunsdon, Mav, Mboverload, McCulley1, Meisam, Michael Hardy, Mindmatrix, Misha Stepanov, Mjchonoles, Mortus Est, Mr Wednesday, Mr.Fortran, Muhandis, Muro de Aguas, NHSAve, Nanshu, NapoliRoma, Nbarth, Neuralwiki, Nghtwlkr, Nikai, Nneoneo, Norm mit, Northumbrian, Ojcit, Okedem, Oleg Alexandrov, Oligomous, Opelio, Ospix, ParallelWolverine, Paul August, Paxsimius, Pgr94, Phil Boswell, Philip Trueman, Pinethicket, Plasynins, Pmcjones, Pnm, Pol098, Poo Yorick, Pthibault, Quadrescence, RTC, Rafiko77, Rangek, Raryl, RaveRaiser, Raysonho, Rechr, Rechingham, RedAndr, RedWolf, Reedy, Rege, Rich Farmbrough, Rigmahroll, Rjwilmis, Roadrunner, RobChafer, Robert Merkel, Robo.mind, Rockear, Rogerd, Rowan Moore, Roy Brumback, Rpropcer, Rravii, Rudnei.cunha, RupertMillard, Ruud Koot, Rwww, SMC, Salih, Saxton, Sbassi, Sccosel, Seba5618, Sequentialis, Skew-t, Sleigh, Smilitude, Smithck0, SnowRaptor, SpaceFlight89, Spacerat3004, Sprachpfleger, Stevenj, Strait, Stsp0269, Stsp0270, Suruena, Svanslyck, T-bonham, TakuyaMurata, Tannin, Tarquin, Taylock, Template

namespace initialisation script, The Cute Philosopher, The Recycling Troll, The Thing That Should Not Be, The morgawr, Thestersatnight, Thom2002, Thumperward, Tiuks, Toke, Tony Sidaway, Tripodics, Twas Now, UnicornTapestry, Urhixidur, Uriyan, User A1, UtherSRG, Van der Hoorn, Van helsing, Van.snyder, Vicarage, VictorAnyakin, Vina, Vssun, Wagers, Wavelength, Wcrose, Wdfarmer, Wernher, Wflong, WhiteOak2006, Who, Wickorama, Widefox, Wiki Wookie, Wikiklrsc, Wikiphyre, Wilkowiki, Williamv1138, Windharp, Woohookitty, Wormholio, Wws, Xamian, Xenfreak, Xlent, YUL89YYZ, Yakushima, Ylai, Yuletide, Yworo, Zap Rowsdower, ZeroOne, Zicraccozian, Zoltar0, Σ, 678 anonymous edits

Lambda calculus *Source:* <http://en.wikipedia.org/w/index.php?oldid=519554462> *Contributors:* 195.166.58.xxx, 2001:4CA0:0:F205:3528:9FC:FD6:4F4A,

2001:4CA0:0:F205:4D67:F88B:4070:3D89, A3 nm, Abcarter, Adamuu, Adrianwn, Aeris-chan, Afarnen, AmigoNico, Ancheta Wis, Andre Engels, AndreasBWagner, AnnaFrance, Antonielly, Apokrif, ArachanoxReal, Ariorivitus, Artem M. Pelenitsyn, Arvindn, AugPi, Awegmann, AxelBoldt, Bbpn, BeastRHIT, Bfries, BiT, Blaisorblade, BlakeStone, Bryan Derksen, Bsmbombdoed, Burritoburrito, CBM, CLW, CLEj37, CYD, CarlHewitt, CeilingCrash, Centrx, Chalst, Charles Matthews, Chbars, Cinayakoshka, Classicaelecon, Cmdrjameson, ComputScientist, Conversion script, Corti, Cyp, D.keenan, DPMulligan, DancingPhilosopher, Daniel.Cardenas, Daniel5Ko, David Gerard, DavisSta, Deattell, Ddidenton, Demosthenes2k8, Derek Ross, Dfletter, Diego Moya, Dominus, Donhalcon, Dougher, Dratman, Dreamyshade, Dreblen, Dtm1234, Dwiddlows, Dysprosia, Edinwiki, Elwikipedista, Enisbayramoglu, EoGuy, Equilibrioception, Eric119, Esap, Evaluist, EvanSeeds, Everyking, Evil Monkey, FalseAxiom, Ffangs, FiP, FubanObfuso, Fuchsias, Furykef, Future ahead, GTBacchus, Gabi1, General Wesc, Giflute, Gioto, Glacialfox, GlassFET, Glenn, GoatGuy, Gploc, Gpvos, GrdScarabe, Grebard, GregorB, Gruu, Hairy Dude, Hakeem.gadi, Hans Adler, Henk Barendregt, Herbee, Hjoab, Hmains, Holedo, Hooperbloob, HowardBGolden, Ht686rg90, Hugo Herbelin, Hyh1048576, Illdimilington, Ilploint, Iridescence, Iridescent, Ironrange, J.Dong820, JLaTondre, JMK, Jacob Finn, James Crippen, JamieVicary, Jan Hidders, Jason Quinn, Jesin, Jimmyzimms, Jleedev, Joel7687, John Baez, John Millikin, John Vandenberg, Jonabney, Jonas, Jonas AGX, Jorend, Joswig, Joy, Jpbown, Jbandes, JulesH, Jyavner, Jyossarian, KHamsun, Kallocain, Karl Dickman, Karun.mahajan, Kba, Keenan Pepper, Kensai, Khukri, Kirk Hilliard, Kizeral, Kliph, Koffieyahoo, Kowey, Krauss, Kripkenstein, Kwantus, LC, LOL, Lambiam, Lance Williams, Landon1980, Largoplazo, Lawandeconomics1, Leibniz, Lexspoon, Linas, Liyang, Lloyd, LoStrangolatore, Longratana, Lotje, Lueckless, LunaticFringe, MBParker, MBIsanz, MH, Makkuro, Malcohol, Maniac18, MarSch, Marcoseda, Markhurd, Marudubshinki, MathKnight, MathMartin, Matt McIrvin, MattGiua, Mattc58, Mcaruso, Mcld, Meiskam, Met501, Mgns51, Michael Hardy, Michael Slone, Minesweeper, Morgaladhi, Mronymil, Mrout, Ms2ger, Msreearhsa, NSiN, Nanshu, Natkuhn, Nczempin, Neelix, Neile, NewEnglandYankee, Noamz, Nomeata, Nonlinear149, Nowhere man, Numi0k, Oleg Alexandrov, OlegAndreev, PJTraill, Paul Richter, PaulAndrews, Pcap, Pedzsan, Pekinensis, PenguiN42, Physis, Piast93, Pinethicket, Pintman, Pvjpjv, Pmranchi, Populus, Potatoswatter, Punto0, Pyrop, Qwertys, Qwpf, R'n'B, R.e.s., Radagast3, Raise exception, Rearete, ResearchRave, Rich Farmbrough, Roaryk, Roccrossi, Royote, Ruud Koot, Sae1962, Sam Pointon, Sam Staton, Sanspear, Sarex, Saric, ScottBurson, Sean ht, Serhei Makarov, Shades79, Sherjilozair, Shredderyn, Siafu, Sligocki, Slipstream, Smack, Softtest123, Some jerk on the Internet, Spiff, Splendor78, Stanmanish, StevenDaryl, Stratocraft, TehKeg, TelecomNut, TenOfAllTrades, The Anome, The Thing That Should Not Be, Theone256, Thryduulf, Thunderforge, Tigrisek, Tim Retout, Timwi, Titoxd, Tobias Bergemann, Tobias Hoevekamp, Tomchiukc, Tony Sidaway, Torc2, Tromp, Trueyourtrueme, TypesGuy1234, UKoch, Ultra two, Urhixidur, VKokielov, Vadmium, Valeria.depava, Vcunat, Vegaswikan, Venullian, Vonkje, Vulture, WLoku, Wavelength, Wedesoft, Wgunther, Whiteknox, WillNess, Wknight94, Workaphobia, WuTheFWasThat, Zaheen, Zero sharp, ZeroOne, Zygmunt lozinski, Пика Пика, 477 anonymous edits

Closure (computer science) *Source:* <http://en.wikipedia.org/w/index.php?oldid=520645224> *Contributors:* 1melquiades, 1exec1, 2chumpy, 2over0, 4th-otaku, AaronSloman, Abcarter, Acehreli, Adavidb, Ajanicj, Akari no ryo, Alberttti, Alexandre Hubert, Alexkon, AllenDowney, Alxndr, Andre Engels, Andrea Kaufmann, Andres, AndrewHowse, Antonielly, Apotheon, Arto B, Asommerh, AxelBoldt, Baguasquirrel, BenFrantzDale, BenKovitz, BenWerb, Bevo, Bhabbit, Black Jam Block, Bluemoose, BonzoESC, Brainsik, C777, Cainetighe, CarlHewitt, Carson Reynolds, Cerberus0, Chadloker, Chowbok, Chris the speller, Chris.dahn, Chrisamaphone, Chturne, Chuunee Baka, Classicaelecon, Cpm, Craig Pemberton, Cubbi, Cybercobra, Dan East, Daniel Earwicker, DejanLekic, Demitsu, Demonkoryu, Denispir, Dermeister, DevastatorIIC, Developer38, Dfletter, Diego Moya, Domenico De Felice, Doug Bell, Dvdrtgn, Ebraminio, Eddie Parker, Edwin175274738, Elaz85, ErikHaugen, Erikdw, Eritain, Eyrain, Fresheneesz, Fubar Obfuso, Furykef, Gafer, Gf uiP, Gordonrox24, Graue, Gretteh, Grokus, Guntaka, Haeleth, Harmil, Hbratt, Het, Hilgerdena, HoodedMan, Hooperbloob, Hugo 87, Indil, InoShiro, Int19h, Ivan Pozdeev, JMCORE, Japanese Seanorin, Jerome Charles Potts, Jerryobject, Jhertel, Jimka, JoeKearney, John Vandenberg, JonathanWakely, Jorend, Jwal, KLee, Karada, Knuton, Kusunose, Lajm, Larry V, LeonardoGREGIANIN, Leothill, Liyanage, Loadmaster, Lulu of the Lotus-Eaters, Madoka, Maduskis, Mankarse, MatthiasRav, Mati22081979, Matt Crypto, MattGiua, Matthew V Ball, Mgsloan, MichaelStanford, Mnemoc, Modify, Mojotoad, Motor, N0mer, Nagle, Nascent, Nathambell, Necklace, NeilFraser, Neile, Netch, NickBush24, Nikhilgupte, Nowhere man, Open4D, Orderud, Orokusaki, OwenBlacker, Paddy3118, Patrickdlogan, Pdelong, Pelago, Pengo, Perfecto, Pgano02, Piet Delport, Pjulien, Plustgarten, Pnm, Poe9514, Pomoxis, Punkgeek, Qduaty, RaulMiller, RedWolf, RedYeti, Reinderien, Renku, Rhubarb, RickBeton, Ripoulet, Robert Pavez, Ron Pinks, Ronburk, Rp, Rtoal, Ruud Koot, SDC, Salix alba, Sam Pointon, Sanders muc, SensuiShinobu1234, Serpex, Shabda, Sheldon Rampton, Sherbrooke, Smallscript, Somerset, Soumyasch, Spoon!, Sun Creator, Svick, TRosenbaum, TakyuMurata, Tbhotch, Technopilgrim, The Anome, TinyFirstman, Tobias Bergemann, Toby Bartels, Tomglomerate, Tony Sidaway, TreyHarris, Uzytkownik, Valentin.samko, VassiliBykov, VictorAnyakin, Warren, Wavelength, WillNess, Wlievens, Yms, Yoric, Zeno Gantner, Zhouzhenghui, 289 anonymous edits

Functional programming *Source:* <http://en.wikipedia.org/w/index.php?oldid=520167280> *Contributors:* 152.163.197.xxx, 203.37.81.xxx, AThing, AV3000, Abasher, Abcarter, Abeliavsky, AceFrahm, Adam Nohejl, Ag, Ahy1, Aij, AlanUS, Alexkon, AliCat, Allan McLines, Amoss, Ancheta Wis, Andrew Eisenberg, Andrewbadr, Angela, Antonielly, Astronautics, AttilaJendvai, AxelBoldt, B Fizz, BD2412, B^4, Bamakharma, BardBloom, Bclarance, Bdesham, Bkil, Blackson, Bobblewik, Brentsmith101, Brian0918, Bubble73, Bunnyhop11, Burn, Burschik, CBM, Cadr, Caminoix, CarlHewitt, Catamorphism, Charles Gaudette, Charles Matthews, Chiswick Chap, Cholling, Chris Roy, Chris the speller, Christiancatchpole, Christopher P. Cjullien, Clements, Coder Dan, Conversion script, Cybercobra, Damian Yerrick, DanDanRevolution, Danakil, Daniel5Ko, David Gerard, Dcourv, Derek Ross, Diego Moya, Diggy Hardy, DiscipleRaynes, Dmerranda, Dominus, Don4of4, Donhalcon, DouglasGreen, Dpol, Dream of Goats, Dtm1234, Dysprosia, Ebmkhfpfc1bkjkpefopeoiaaggioah, Einstrek, Elias, Elwikipedista, Euchiasmus, Ezrakility, FatalError, Fbartolom, Firesofmayn, Flex, Flohsuchtliebe, Flurble, Franci Lima, Frencheigh, Fubar Obfuso, Fuchsias, Functional penguin, Functionalguy, Gaius Cornelius, Gajakannan, Gazpacho, Gerweck, Giftite, GoatingB, Goofyheadedpunk, Gpvos, Graham87, GregorB, Gwern, Hamaryns, Hooperbloob, Ida Shaw, Ideogram, Igouy, Ironcamel, Isarra, Itchy Archibald, Julian, Ivan Štambuk, J'axis, J.delaney, JBSupreme, JC Chu, JRocketeer, Jeevanpingali, Jeoffw, Jericho4.0, Jerryobject, Jhknight, Jkelly, Jmha, Jonsafari, Jpbown, Jpkotta, JulesH, JustinWick, Justinc, Karada, Karl Dickman, Kate, Kelly Martin, Kenny sh, Kerrick Staley, Ketiltrout, Kozuch, Kurieeto, LOL, LauBjensen, Leibniz, Lerollec, Levin, Lexor, Liberatus, LinguistAtLarge, LogaRhythm, Lulu of the Lotus-Eaters, Lupin, M gol, MONGO, MPeller, MSully4321, Magnhus, Magioladitis, Mahir256, Malleus Faturorum, Mangst, MartinRinehart, MattGiua, MattTait, Matthias.f, Matusz, Mboverload, Mdkirkf, Mdmcginn, Meanskeep, MementoVivere, Michael Hardy, MichaelSpeer, Mild Hiccup, Minesweeper, Mjs, Mudphone, Mxvi, Nbarth, Neight108, Neile, Nighl Gyr, Ninly, Nixavar, Objec01, Obradovic Goran, Offenbach, OldCoder, OlivierBoudeville at EDF, Orbekk, Palfrey, Pamplune, Paulgush, Pgano002, Philip Trueman, PhilipMW, Philopedia, Physis, PieterKoopman, Ptrb, Puckly, Qwertys, RHaden, RTC, Raboof, Raise exception, Rajakhr, RaulMiller, RaveX, Rebooted, RedWolf, Reinderien, ReiniUrban, Requestion, ReyBrujo, Rich Farmbrough, Richard W.M. Jones, Riedl, Rjwilmis, Rlevse, Rodney Boyd, Ruud Koot, Röni, RScominator, Sanjay742, Sapeur, Sbhomm, SebastianHelm, Semi Virgil, Sfan00 IMG, Shime, Shipmaster, Simoneau, Sithy, Slaniel, Sloose, Snoyes, Spyارد, Ssd, StewartMH, Strat, Sun Creator, Sunyin, TakuyaMurata, Taw, Tejoka, Tezh, Thadius856, The Anome, The Cunctator, TheNightFly, TheSeven, Theflyngman, Thehebrehammer, Thomasthep, Thumperward, Timhoooy, Timlevine, TitoAssini, Tjdw, Tobias Bergemann, Tobias Hoevekamp, Toddst1, Tokyotx, Tom.lanning, Tonypiazza, Toussaint, Trappist the monk, TravisSwicegood, Twobitsprte, UKWikiGuy, Uzytkownik, VKokielov, Vanished user 5zariu3jisj04irj, Varuna, VeryVerily, VictorAnyakin, Vocaro, Wadler, Wavelength, WhatamIdoing, Whiteknox, WillNess, Winston Chuen-Shih Yang, Xmlizer, Ycor, Ysangkok, Zoicon5, Zoltar0, احمد, 445 anonymous edits

Lisp (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=520271420> *Contributors:* (-Barry-, 15dancing, 217.70.229.xxx, 4.54.210.xxx, 4hodmt, A.Ward, AApathy, AHMartin, Abcarter, Adam majewski, Adriatikus, Aguerodc, Akesi, Aldarione, Alex Heinz, AlastairMcMillan, Altasoul, Ancheta Wis, Andre Engels, Andres, Anthony Appleyard, Antonzoni, Ap, Aranel, ArneBab, Arnehanas, Arto B, Atlant, Azzard, BD2412, Barak, BartonM, Batkins, Bdesham, Beinsane, BenBaker, Betruneker Affe, Bevo, BiT, BlakeStone, BobdC, Boemanneke, Boshomi, Brian Kendig, Bricevdm, Brion VIBBER, Broquaint, Bryan Derksen, Bryanmackinnon, Bstarlyn, Buguldey, Burzmali, CRGreathouse, CUSENZA Mario, CYD, Can't sleep, clown will eat me, CanisRufus, CarlHewitt, Cassowary, Ccreitz, Chaos5023, Charles Gaudette, CharlotteWebb, Chickencha, Chowbok, Chris Roy, Chris the speller, Chris-gore, Chroxildyphitic, Chuck369, Chuckhoffmann, Closedmouth, Coatonasuanas, Collabi, Conversion script, Creidieki, Crownest, Cstacy, Curly Turkey, Curps, Cybercobra, DGerman, DNewhall, Damian Yerrick, Danakil, Daniel Brockman, Danim, Daveh1, David.Mestel, David.Neves, Dcoetzee, Deflective, Dekails, DekuDekuplex, Denkoryoku, Derek Ross, DerikSmartVsTheIncredibleCokeMachine, Dicklyon, DiscipleRaynes, Disque71, DocWatson42, Dpbsmith, Draco, Dragentsheets, Dreamyshape, Drinies, Ds13, Dustin gayler, Dysprosia, Dzhebi, Dzlk, Eedwa 01, Ehn, Elimisteve, Ellmist, Eloquence, Epr123, Erict, Erik Sandberg, Ewlyahoocom, FFG-31, Fagstein, Faisal.akeel, Famouslongago, Fbjon, Feezo, Ffangs, Figs, Forlornturtle, Fran McCrorv, Frank Gearlings, Frank Shear, Fred Birchmore, Fredrik_FrenchsAwesome, Fromagesteel, Froth, Fubar Obfuso, Furykef, Gadrium, Gail, Garzo, Gazpacho, Giot, Gen Pecipelli, Glrx, Gmilza, GoingBatty, Goodnightmush, Gorbag2, Graham87, Graue, GregorB, Gronky, Groog, Grue, Gwern, Hackergene, Hairy Dude, HeikoEvermann, Hgb asicwizard, Hirzel, Hom separta, Houseofhak, Hrungnir, Iamscked, Ifomichev, Igottalisp, ImMAW, Imran, InverseHypercube, Inzy, Iterator12n, Ivan Štambuk, J128, JMSwtk, JW1805, James Crippen, JasticE, Jehochman, Jensgb, Jerryobject, JidGom, JimVC3, John254, JorgePeixoto, Jorm, Joswig, Jotabede, Jpkotta, JuJube, Jwalling, Jzw, KKong, Karl-Henner, Katieh5584, Kb, Kemiv, Kenyon, KevinLayer, Klimov, Kingspook, Koepflinger, Koyaanis Qatsi, Ks2772, Kvaks, Kwamikagami, Kwertii, LOL, LauriO, Levin, Lightmouse, Linas, Lkesteloot, Loadmaster, M3tainfo, M4gnumOn, MC10, MER-C, MIT Trekkie, MITAlum, Macrakis, Mandara, MarSch, Margin1522, Marudubshinki, Maskedycodeler, Maslin, Maurobio, Maximax, Melaen, Miguel, Mike Rosot, Mikera, MilesMi, Mindmatrix, Minesweeper, Moe Aboulkheir, MrDolomite, MrBll, Mrmathematica, Mschlindwein, Mykhkal, Nalvage, Nanshu, Neil Tallim, Neile, Nentuaby, Netfed, Netsettler, Nick Levine, Nilboy, Ningauble, Nixeagle, Obscuranym, Ojw, OrgaG, Orderud, OrgasGirl, Orthogonal, P0lyglut, Patche99z, Paul Ebermann, Paul Stansifer, Pauli133, Pi zero, Piet Delport, Piggy@babaqi.chi.il.us, Piloguy, Piquan, Pmcjones, Pne, Polypus74, Poor Yorick, Q Linx, Qbg, Quackers1, Qwertys, Qwpf, Raith Preston, RedWolf, Reedy, ReinUrban, Renesis, Rfc1394, Rglovejoy, Rich Farmbrough, RickScott, Rlw, Robo.mind, Robost, Ross Burgess, Ruud Koot, Rxfelix, Sae1962, Salvatore Poier, SamuelTheGhost, Sarken, Sburke, Schol-R-LEA, Seidenstud, Sergey Dmitriev, Sgwfmk8, Shafei, Shii, Shizhao, Shortgeek, SidP, Siggylama, SingingWolfboy, Slady, Smack, Smilack, Smyth, Solrie, Bizna, Somercet, Spdegabrielle, Sridharinfinity, Stan Shebs, Stassats, Stewartadcock, Stratocracy, Stubblyhead, Svick, Swiftly, Ta bu shi da yu, Tacitus Prime, Tagus, Taw, Template namespace initialisation script, The Anome, The PIPE, The Wednesday Island, The wub, TheMightyOrb, Thom2729, Thomas Larsen, Tide rolls, Tinku99, Tobby72, Tobias Bergemann, Tompsc, Tony Sidaway, Tordek ar, ToreN, Torla42, Townson2003, Transysfaiz, Traroth, Trustable, Tunnable, Twimoki, Twinxor, UtherSRG, Val42, Velza, Vfp15, VictorAnyakin, Vvk2991, WMarsh, WOSlinker, WatchAndObserve, Wavelength, Who, Wickorama, Wiki Wookie, Wikiborg, Wikiklrsc, Wile E. Heresiarch, Woohookitty, Wpac5, Wws, Xxovercastxx, Yahoolian, Yakushima, Yath, Yonkeltron, Yunner, Yworo, Zapraki, Zickzack, Ævar Arnfjörð Bjarmason, 612 anonymous edits

Pattern matching *Source:* <http://en.wikipedia.org/w/index.php?oldid=520564220> *Contributors:* Ahoerstemeier, Amr.rs, Andreas Kaufmann, Arno Matthias, Ash211, AxelBoldt, CSWarren, Canterbury Tail, Classicaelecon, D, Deewiant, Derek.farn, Donald Albury, Dougher, DutchDevil, Dysprosia, Genefects, GhettoBlaster, Giflile, Gioto, Grafen, Greendr, Gwern, Hazelorb, Jeff3000, John of Reading, JonHarder, Karl.redman, Liuxiaoming, Magioladitis, Marudubshinki, Nbarth, Neile, Ojigiri, Op47, P0lyglut, Parallelized, PyroPi, Pytchblend, Qu3a, Quadrescence, Requestion, Rich Farmbrough, Robinjam, Roman Munich, Rorro, Ruud Koot, Sam Pointon, Scope creep, Scorchsaber, Shermanmonroe, Simeon, Spayard, Sun Creator, TonyMath, Tuukkah, Vcunat, Verdatum, Wikipediast, Wtmitchell, 104 anonymous edits

Anonymous function *Source:* <http://en.wikipedia.org/w/index.php?oldid=520599345> *Contributors:* 1ForTheMoney, Abednigo, ArchiSchmedes, Arto B, Bakkedal, Bakken, Benizi, Benwing, BiT, Bigtrick, Brighterorange, Brvman, Btx40, Bunnyhop11, CRGreathouse, Churnett, Cebarber, Cmglee, D.Lazard, Delphinus1, Devile, Dmon137, Dougher, Dtm1234, Edward, Eivindw, Eregonp, Erudecor, Fluoborate, Flomyt, GhettoBlaster, Het, Hydrox, IThinkTheseUsernamesAreStupid, JC Chu, JLaTondre, Jclatlin, Jerryobject, Joeythehobo, Kazvorpal, Ketil, Kobi L, Korval, Masonzilla, Metalboy5150, Michael.Hardy, Mild Bill Hiccup, Mistman123, Modify, Moonwolf14, Mormegil, Mortense, Myna vajha, NedGladstone, PMLawrence, Pcap, Pntori, Psychotic Spoon, Ptery, Redhanker, Renku, RokerHRO, Rowandavies, Roybristow, Ruud Koot, Sae1962, Scientus, Searke, Sjiver, Sligocki, Soimort, Spoon!, Squeek, Surturz, Thardas, The Thing That Should Not Be, Thomas Linder Puls, Tobias Bergemann, Trustable, Uzume, Vikrant42, Wavelength, Whitepaw, Winston Chuen-Shih Yang, XP1, Xavier Combelle, Xcvista, ^zer0d3r\$, Δ, Αχ.α, Σ, 風狼翁, 168 anonymous edits

Scheme (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=520827242> *Contributors:* -Barry., 195.149.37.xxx, 212.134.22.xxx, Abdulaziz Ghuloum, Acaides, Al E., AlexG, Alphax, Amir.h.110, Amrit, AnOddName, Angus Lepper, AnonMoos, Ap, ArglebargleIV, Armeth, Ashawley, Aubrey Jaifer, AxelBoldt, Azaostro, B Fizz, Barak, Bdosrror, BenKovitz, Bkl, Blaisorblade, BlakeStone, BlueBobbis, Bobde, Boemanneke, Boshomi, Bovineone, Brianjfox, Brighterorange, Burschik, Cadr, CanisRufus, CarlHewitt, Cat Parade, Churnett, Chalst, Chaos5023, Chowbok, Chridd, Chrisdone, Christian List, Chzz, Cmbranone, CommKing, Conversion script, Cornflake pirate, Creidieki, Ctdean, Cww, Cybercobra, Damian Yerrick, Danakil, Danilkutkevich, David Gerard, DavidCary, Davidpk212, Dcoetze, Delldot, Dftyapo, Dicklyon, Digitalq, Direwolf, Dlambert, DmWi, Donhalcon, Dipolomansion, Druffic, Dvanhorn, Dysprosia, Edward.in, Edmonton, Egil, Elitarizay, EngineerScotty, Equentidit, Eric119, Erkcan, Espen, Esrogs, Eupraxia, EvanProdromou, Excallestis, Ffangs, Firsfron, Flerhun, Fredrik, Fubar Obfusco, Furykef, Gadflum, Gaius Cornelius, Galoubet, Gary King, Gdr, Gf uiP, GhettoBlaster, Ghoseb, Goetter, GreenUpGreenOut, GregorG, Gro-Tsen, Grshiplett, Grzegorz, Gwern, Haham hanuka, Hananoshi, Handicapper, Hans Oesterholz, Harald Hansen, Hephaestus, Herbee, Hga, Hojimachong, Homerjag, Hongooi, Huffers, Huge colin, Iamfsccked, Iamtheari, Ian Clelland, Ideogram, Indeyets, Intangir, Integral creature, Iopq, Isaac Rabinovitch, Ivan Štambuk, Ixf6d4, J.delanoy, JLaTondre, JVz, Jarble, Jc helary, Jfmantis, JohnKozak, Joswig, Jpvinal, K1ngyo, KSmrq, Kanjy, Katiel5584, KbK, Kenyon, Kertrats, Kingpin13, Knepfleler, Kusunose, Kuszi, Lapax, Larry V, Legend124, Lentower, Letdinosaurusde, Levin, Ling.Nut, Lobner, Lulu of the Lotus-Eaters, MH, MIT Trekkie, Mag.v, Male1979, MarSch, MartinHarper, Marudubshinki, MattGiua, Matthias.f, Mav, Mbh, Mcses, Memotype, Mhh191, Mikael.more, Mikaka, Mikeblas, Mikko Paanamen, Morte, Motor, MrMathematica, Myrfd cb, Nandesuka, Nanshu, NationalSpace, Nczempin, Neile, Nikai, Njyoder, No-body, Nohat, Norm mit, Not-just-yet, Officiallyover, Ogranut, Ojw, Ozten, P Nutz H4x0r, P0lyglut, PTP2009, Paradoxsociety, Pavel Vozenilek, Physicistjedi, Piet Delport, Pilotguy, Pomte, Poor Yorick, Pps, Pyrotec, Quamaretto, Qutezuce, Qwertys, R2q2, R3m0t, Raingrove, Rajakhr, Reisio, Rijkbenik, Rjwilmsi, Romann, Rror, Rurus, Ruud Koot, Saintamh, Salasks, Saligrone, SamB, Samohyl Jan, Sdfisher, SeanProctor, SergeyLitvinov, Shaydon, Sibian, Slashem, Smack, Somercet, Soumyasch, Spdegabrielle, Spikey, Spoon!, Squirmymcphee, Stannered, Stassats, StelioKlm, Stephan Schulz, StevenJ, Stuhaking, Sun Creator, Sunnan, Takikawa, TakuuyaMurata, Tblklass222, Ted-gao, Teddiego, Tehom2000, Template namespace initialisation script, Thunderbolt16, Tikiwont, Timickey, Timwi, Tizio, Tobias Bergemann, Tony Sidaway, Tony1, Traroth, TraxPlayer, Tuahla, Unforgettableid, UserGoogol, UtherSRG, VeryVerily, Voidxor, Wavelength, Weel, WelshMatt, Wernher, Wickorama, Wlievens, Xyb, Yym, Z0ltanz0ltan, Zaheen, Zeno Gantner, Zoechi, Zsam, Σ, 401 anonymous edits

Type inference *Source:* <http://en.wikipedia.org/w/index.php?oldid=512673630> *Contributors:* 786b6364, Aardvark92, Adrianwn, Alansohn, Almkglor, Ancheta Wis, Ascánder, AxIrosen, B4hand, ChuckEsterbrook, Classicaelecon, Clegoues, Cobalt pen, Connelly, Cybercobra, Damian Yerrick, Daniel5Ko, Debajit, Dogcow, Dysprosia, Episteme-jp, ErikHaugen, Euyyn, EvanED, Francis Lima, Gaius Cornelius, Gasper.azman, Gf uiP, Gfxmonk, GhettoBlaster, GoingBatty, Gregbard, Isaadealey, Jabowery, Jackson, Jerryobject, Jhammerb, Jonathan S. Shapiro, Koper, Kyralessa, Leonard G., LinkTiger, LittleDan, Ljaun, MarXidad, Marudubshinki, Mgreenbe, MrBlueSky, Neile, Nighthawk4211, Oerjan, Owengibbins, Peni, ProjectSHiNKiROU, R. S. Shaw, Rjwilmsi, Ruakh, Ruud Koot, Sae1962, SamuelRiv, Semi Virgil, SparsityProblem, Spayard, Talandor, That Guy, From That Show!, Tobias Bergemann, Torc2, Zoicon5, Денис Владимирович, 115 anonymous edits

Currying *Source:* <http://en.wikipedia.org/w/index.php?oldid=520572938> *Contributors:* 16@r, AaronSloman, Amakuru, Ar, AxelBoldt, BGOATDoughnut, Baffo, Beland, Belg4mit, Bkkbrad, Bobo192, Bonotake, Brighterorange, CBM, Caa4, Carewolf, Cdiggins, Chalst, Classicaelecon, Codewritinfol, Conversion script, Cybercobra, Damian Yerrick, Daniel5Ko, Dastle, DenisDollfus, Dlinsin, Dominus, Dougnukem, Dysprosia, E is for Ian, Easys12c, Ehrd, Eprb123, Esden999, Flemira, Flesler, Fresheneesz, Frungi, Fuchsias, Garde, Gazpacho, Gdr, Giese, Giftlite, Graham87, GregorB, Hairy Dude, Iliu Kr., Inkeddmn, JVz, Jaimic, Jason Quinn, Jboden1517, Jdh30, Jefallbright, Jeronimo, Jessecurry, Jimw338, JoeKearney, John Vandenberg, Kigor k, Kallocain, Karl Dickman, Knotjl, Lambiaria, Langec, Linas, LittleSmall, Lmgottschalk, M4gnun0n, Macrakis, Malcolm Rowe, Marvanzee, Marudubshinki, Matt Crypto, Mellum, Mets501, Mhss, MichaK, Michael.Hardy, Minesweeper, Mr Taz, Mtanti, Neile, NevilleDNZ, Noczes, Nofxjunkie, Oising, Paddy3118, Patrick, Pcap, Pdhooper, Piet Delport, Polypus74, Qartis, Qwertys, Rjwilmsi, Rob.desbois, Roger Rohrbach, Rswarbrick, Ruud Koot, Sctgrp, Senu, Shai-kun, Spayard, Subsolar, TakuuyaMurata, The Anome, Thumperward, Timwi, Tirerim, Tjo, TomyDuby, Tony Sidaway, Trontonic, Tuukkah, Tyler-willard, Uncopy, Unfree, Violetriga, WPWoodr, Wdevald, Wgunther, Wikipelli, Yanroy, Zootm, 156 anonymous edits

ML (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=520145682> *Contributors:* AHMartin, AaronNGray, Apokrif, Ascánder, Baguasquirrel, Barkeep, Bdesham, Blaisorblade, Bowman, Brentsmith101, Chrisahn, Christopher Monsanto, Cjhonsone, Conversion script, Cwoolfsheep, Cybercobra, Cyneb, Danakil, Daniel5Ko, Danim, Dantheman, David Eppstein, David.Wahl, Dcoetze, Delirium, Derek Ross, Diego Moya, Djspiewak, Donhalcon, Dysprosia, Edcolins, Elaz85, Emperorbina, Ferengi, Fuzlogic, FvdP, Getsecure, Greendr, Gwern, Haeleth, Harryboyles, Hetchel, Ipsign, Irrbliss, Isnow, IvanStambuk, JohnBates, Jonik, Joriki, Jnsx, Lagelspeil, Lotje, Mahr126, Malekith, Markawika, Masharabivovich, MastCell, MattGiua, MementoVivere, Metromoxie, Michaeln, Minesweeper.007, Norm mit, Obscuranym, Offensive, confusing, or unreadable text or characters, Osmodiar, Oxymoron83, PGSONIC, Pejman47, Pne, Poor.Yorick, Positron, Qwertys, Rbonvall, Rmaus, Ruud Koot, Ryan94114, Sae1962, Samsara, Schapel, SeanProctor, Senu, Sestoft, Shigoel, Spacemartin, Spoon!, SteveLoughran, Stilroc, Tagus, The Wild Falcon, Throwaway85, Timwi, Tobias Bergemann, VeryVerily, Vlops, Vrable, Wavelength, Wickorama, Wiml, Yworo, Zecc, 100 anonymous edits

Standard ML *Source:* <http://en.wikipedia.org/w/index.php?oldid=519003815> *Contributors:* AaronSloman, Adrianwn, Alksentr, AnAj, Awkwardusername, Azmi1995, Brighterorange, BurntSky, Cek, Chaoticneuron, Chinju, ChrisKalt, Cjoev, Cwolfsheep, Cybercobra, Cyneb, DNewhall, Danakil, David.Monnias, Dcoetze, Delirium, Donhalcon, Edward, Eliotyork, Emperorbina, Fughgettaboutit, Greenrd, Gribra2010, Halladba, Iridescent, Jkl, Joriki, Jrcinayc, Jrtorhe, KnightRider, LOL, Leibniz, Leonardo Lang, Michaeln, Midinasturazz, Minesweeper, Mpiesenbr, Neile, Nick Barnes, OriumX, Ornil, Pavel Vozenilek, Pne, Pomte, Qwertys, R'n'B, Rebooted, Rockfang, Ruud Koot, Ryan Norton, Sboriah, Sebbe, Sestoft, Stephen Bain, Stewartadcock, Sun Creator, Supersofts, The Wild Falcon, Thue, Thumperward, Tobias Bergemann, Urhixidur, VeryVerily, VictorAnyakin, Vukuncak, Vofly, Vrable, Wiml, Zarvak, ZeWrestler, 146 anonymous edits

Tail call *Source:* <http://en.wikipedia.org/w/index.php?oldid=511824667> *Contributors:* Andreas Kaufmann, Bertport, Blaisorblade, DFRussia, Damian Yerrick, Gpvos, Janto, Jkl, Kotika98, Krinkle, Kwi, Liberatus, Lightmouse, LilHelpa, LinguistAtLarge, MattGiua, PMLawrence, Pgr94, Royote, Ruakh, Ruud Koot, Rywebb, Saraphim, Tobias Bergemann, Tony Sidaway, Wapcaplet, WillNess, Wprlh, 36 anonymous edits

Lazy evaluation *Source:* <http://en.wikipedia.org/w/index.php?oldid=519902592> *Contributors:* 16@r, 213.253.39.xxx, Aaronbrick, Abcarter, Adamrmos, AgadaUrbanit, Alex.atkins, Amelio Vázquez, Andreas Kaufmann, AndreasFuchs, Blaisorblade, Bobrayner, BrentDad, Bryan Derksen, Btx40, Catamorphism, Caeu.cm.rego, Centrx, Chrylis, Cjgiirdhar, Conversion script, Cybercobra, Danielx, David.Wahl, David-Sarah Hopwood, Derek Ross, Donhalcon, DouglasGreen, Dysprosia, Ed.Poor, Ed g2s, Edward, Fanf, Favonian, Furykef, Gerweek, Gf uiP, GhettoBlaster, GoingBatty, Greenrd, Guy, Gwern, Hairy Dude, Hans.Dunkelberg, IamOz, Isnow, J. Finkelstein, JClately, Jboden1517, Jerryobject, Jluscher, Kimbly, Kwertii, LOL, Levin, Locos epraix, Lulu of the Lotus-Eaters, M4gnun0n, Magnhus, Magnus Manske, Marudubshinki, Maxim Razin, Mfloryan, Michael Sloane, MichaelJanich, Mpkr, Ms2ger, Newman9997, Nibios, NickyMcLean, Njsg, Nneonneo, NotASpammer, Only2sea, Pablo X, PatrickFisher, Paul Richter, Pcb21, Perelaar, PeterJanRoes, PeterJordens, Phil Boswell, Pinar, PlaysWithLife, Ppl, Pps, Project2501a, Quintus314, RHaden, Recluse wiki, RedWolf, Rich Farmbrough, Richard W.M. Jones, RichardF, Robert Merkel, Robykiwi, Rycle731, Ruakh, Ruud Koot, SilkTork, Simeon, Spoon!, Steven Jones, Suruena, TakuuyaMurata, Tarquin, Tassedethe, Taw, The Anome, Tobias Bergemann, TomWij, Urhixidur, Vkokielov, VrmIguy, Wavelength, WelshMatt, Ødipus sic, 84 anonymous edits

Haskell (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=519954265> *Contributors:* 134.129.58.xxx, 144.132.75.xxx, 1exec1, 2001:600:5003:410:E1F4:8DCF:F078:1436, 64.105.112.xxx, 66.68.81.xxx, 7, Aaron McDaid, Adam Blinkinsop, Adrian.benko, Ajk, Akitstika, Alai, Algae, Alro, Ancheta Wis, Andy.melnikov, Angrytoast, Angu Lepper, Are you ready for IPv6?, Arjun G. Menon, Armin.Rigo, Ary29, Ashley.Y, Autrijus, Axman6, B4hand, Bardo zero, BenRG, Benja22, Benjamin.Barenblat, BiT, Billgordon1099, Bkell, Black Falcon, Blanford robinson, Bleakgadfly, Bobdc, Boemanneke, Brian0918, Brion VIBBER, CRGreathouse, Cadr, Calmer Waters, Canderia, CapitalSasha, Catamorphism, Cdamama, Chief sequoya, Chris Roy, Chrsdone, Cic, Clconway, Cnilep, Coffee2theorems, Constantine, Conversion script, Crackerjack, Cryoboy, CryptoDerk, Cybercobra, Daf, Danakil, Daniel5Ko, DanielPharos, Dcoetze, Deflective, Delirium, Deveshprabhu, Dewi78, Dikke poes, Dominus, Donkeydonkeydonkey, Dons00, Dougher, Doulos Christos, Dysprosia, Eklitzke, Erik Sandberg, Errnmedia, Et764, Ettrig, Eupraxia, Evice, Flesler, Flugschtfie, Forejt, Frakturfreund, Fredrik, Fresheneesz, Fubar Obfusco, George Laparis Funk Studio, Gianluigi Tegazzi, Graham87, Graue, Greenrd, Gribra2010, Gwern, HV, Hairy Dude, Henning.Thielemann, HenriqueFerreiro, Hextad8, Hinrik, Igouy, Imz, InverseHypercube, Ithika, Ivan Štambuk, JLaTondre, Jaahee0113, Jarble, Jboden1517, Jbwhtmore, Jeff G., Jericho4.0, Jerryobject, Jesin, Jmanson, John Millikin, JohnM, Jonasfagundes, Jonpro, Jrosdahl, Jrw1020, Jufert, K.lee, Karada, Keenman76, Ketil, KnightRider, Kowey, Kuszi, Kwamikagami, LOL, Lacen, Larry Hastings, Leskhal, Levin, LittleDan, Lordmetroid, Lycurgus, M4dc4p, MH, Malleus.Fatuorum, Mani1, MarSch, Marasmusine, Marianocecowski, Marudubshinki, Marx Gomes, Matt Crypto, MattTait, Maximus Rex, Mcc.ricardo, Mczack26, Meewam, Melnakeeb, Mgdedmin, Michael.Hardy, Mikesteel81, Mild Bill Hiccup, Mindmatrix, Minimiscience, Miym, Mormegil, Mr. Lefty, Mr.Ollie, Msg555, Nghtwlkr, Obscuranym, Oerjan, Okmo, OllieFury, Ortolan88, Osmodiar, Palpalpalpal, Pavel Vozenilek, Paxcoder, Peterhi, Pgan002, Phillyidol, Piet Delport, Pingswept, Polyparadigm, Polypus74, Poor.Yorick, PureJadeKid, Qutezuce, Qwertys, R'n'B, Rajakhr,

Ramsey Smith, RandomStringOfCharacters, Reinis, Renku, Richardcavell, Richardmstallman, Rookkey, Ruakh, Rudd Koot, Rwl4, Salix alba, Sam Hocevar, SamB, SamuelTheGhost, Sandos, Sathiyamoorthyp, Sbabra, Scott5114, Scravy, Semifinalist, ShelfSkewed, Shoejar, Shreders, Shreevatsa, SimenH, Simeon, Simonmar, Slipstream, Sméagol136, Snaxe920, SoLando, Soumyasch, Squids and Chips, Staeker, StormWillLaugh, Strcat, Svick, Tagus, Taktoa, Taw, Tbhotch, Tejoka, The Letter J, TheCoffee, Themfromspace, ThomasOwens, Timwi, TittoAssini, Tizio, Tobias Bergemann, TomLokhorst, Tomaxer, Torc2, Traroth, TuukkaH, TwoOneTwo, Utorsch, VeryVerily, VictorAnyakin, Vindicator26, Voidxor, Walkie, Wanders, WatchAndObserve, Wernher, Whatgoodisaroad, Wickorama, WillNess, Willking1979, Winterborn, Wwwwolf, X-G, Xan2, Xillimiandus, Xodarap00, Zfr, 453 anonymous edits

Type system *Source:* <http://en.wikipedia.org/w/index.php?oldid=519849798> *Contributors:* 121a0012, 1exec1, A3 nm, Aaron Rotenberg, Administration, AdrianLozano, Adrianwn, Agarwal1975, Ahy1, Aleksd, Allan McInnes, AllenDowney, Altenmann, Alterego, AnAbsolutelyOriginalUsername42, AnAj, Ancheta Wis, AngryBear, Anshee, Antonielly, Anuroop Sirothia, Anwar saadat, Ash211, Atreyu42, Audriusa, AutumnSnow, AvicAWB, Barabum, Beland, Blaissorblade, Bluemoose, Bob O'Bob, Bubb3, Caesura, Carlosayam, Cartiolt, CheesyPuffs144, ChorizoLasagna, Chridd, CiudadanoGlobal, Cjoev, Classicalcon, Clements, Cntras, Comatose21, Connelly, Cpiral, Craigbeverage, Cybercobra, Damian Yerrick, Danakil, Daniel-Dane, Daniel.langdon, Darxus, David Nicoson, David-Sarah Hopwood, Dcoetzee, Denny, DiscipleRaynes, Donhalcon, Doradus, Dougher, DouglasGreen, Dpv, Dreftymac, Droob, Drumheller, Dysprosia, Edaelon, Edward, Elaz85, Elliot.jaffe, Emperorbma, EngineerScotty, Ennu93, Epolk, Erc, Eric119, Esap, Eugeneiun, Euyyn, Exigentsky, Ezrakilty, Feis-Kontrol, Fieldmethods, Fooblizzo, Foxygirlnamara, Fredrik, Fubar Obfuscous, Furby100, Furykef, GB fan, Gail, Gdr, Georg Peter, Gerwerk, Gf uip, Ghettoblaster, Giftlife, Gimlk, Gracenotes, Grandscribe, Grauenwolf, Gwern, Hairy Dude, Hammer, Harmil, Hdante, Headbomb, Hippietrail, Hmlaptops, Iggywmangi, I hope127, Imperator3733, Irizedine, JIP, JLaTondre, Jason5ayers, Javawizard, Jbolden1517, Jeargle, Jeban Jef-Infoje, Jen savage, Jeronimo, Jerryobject, Jfire, Jleedev, John lindgren, JonHarder, Jopinaca, Joswig, Jtayloriv, Julesd, K.lee, Karl Dickman, Karouri, Kbdenk71, Kbrose, Ken Gallager, Ketil, Klondike, Krischik, LOL, Langee, Larry V, Lawpj, Leibniz, LittleDan, LoStrangolatore, Lowellian, Lulu of the Lotus-Eaters, Maclary, Mange01, MarXidad, MarLevel3, Mark Renier, Marksilkibeck, Martin Hampel, Martinskibshki, MattGiua, MattOConnor, Maximaximax, Maximilianklein, Mfc, Michael Sloane, Michal Juros, Mikeblas, Mikon, Minesweeper, Mjamja, Mjy, Mmdoogie, Mmernex, Mnduong, Mokhov, Moonwolf14, MrBlueSky, Msonle, NHSavage, Nabla, Neile, Nightstallion, Norm mit, Norman Ramsey, Nuno Tavares, OriumX, P00r, Paddy3118, Palmluster, Patrick, Paul Richter, Pcap, Pedant17, Peepedie, Pengo, Peterjedon, Pfeilsitzpe, Phil Boswell, Phorgan1, Pit, Pjb3, Pomoxis, Poor Yoric, Prodego, Qwfp, R. S. Shaw, Raise exception, RandalSchwartz, RedWolf, Reinderien, Riley Huntley, Rjwilmsi, Robykiwi, Rogper, Rookkey, Ross Fraser, Roybristow, Ruud Koot, SLi, SadaraX, Sae1962, Sagaciousus, Scemerlin, Selioiou, Serval333, Simeon, Simetrical, Simoes, SimonP, Simoneau, Slaniel, Smeatish, Snyoes, Stevenj, Strake, Sun Creator, Sureuna, Swift, Sykopomp, Tablizer, TakuyaMurata, Tasc, That Guy, From That Show!, The Anome, TheNightFly, TheProgrammer, Therog1, Thincat, Thumperward, Tim Starling, Tim Watson, Tobias Bergemann, TomStuart, Torc2, TuukkaH, Twilsonb, Updatetbjarni, Urhixidur, VampWillow, VictorAnyakin, VladimirReshetnikov, Washi, Wavelength, WhiteCat, Wjhonsom, Wlievens, Wrp103, Ww, YahoKa, Yahya Abdal-Aziz, Yoric, Ysoldak, Zron, Σ, 378 anonymous edits

Scope (computer science) *Source:* <http://en.wikipedia.org/w/index.php?oldid=520721045> *Contributors:* 28bytes, Abdull, AdamCox9, AlanUS, Aleksander.adamowski, AlexanderTsamutali, Alfio, Avriette, Banshee, BiT, BrideOfKripkenstein, Brighterorange, Btx40, CYD, CalPaterson, Ccommett, Cesarb, Christopherlin, Chturne, Cjoev, Cniggeler, Cybercobra, Denis bider, Diego Moya, Djib, Dysprosia, Eelis.net, FatalError, Ferris37, Fredrik, Gdevlugt, Geary, Glenn Willen, Godtviken, Husky, Hyju, Iogard, Infovarius, Isidore, JLaTondre, Jacobolus, Jakevoytko, Jbolden1517, Jerome Charles Potts, Jerryobject, Jni, Joey Novak, JonHarder, Jtayloriv, KazKylheku, Kbdank71, Kcordina, Kshieh, Kusunose, LOL, Leushenko, Marc omorain, Marchash, Mark Foskey, Martychen, MattGiua, Mechanical digger, Michael Hardy, Mkoval, Mwarren us, NYKevin, Nickschu, Onirivoner, Patrick, Payrad, Pfeldman, Pointillist, Rbonvall, RedWordSmith, Rfc1394, Rhamer, Ricardo Ferreira de Oliveira, Rigelan, Rp, Ruaku, RucasHost, Ruud Koot, Slamb, Spoon!, Ssd, Subversive.sound, Sunny Sachanandani, Svick, TakuyaMurata, TheThomas, Tlesher, Tobias Bergemann, Wavelength, Wayne Slam, Wernher, WillNess, William M. Connolley, William spurlin, Wiremore, WorldsApart, 100 anonymous edits

Class (computer programming) *Source:* <http://en.wikipedia.org/w/index.php?oldid=519294106> *Contributors:* ., 4321dcba, Abdull, Ahkilinc, Ahmad510, Aknowl, Alan D, Alan012, Alan_D, Amakuru, Antonielly, Anupamsr, Assulted Peanut, Avicennas, Batpox, Ben.d.zimmer, Betacommand, Blaxthos, Bluemoose, Bmf1972, Bobo192, Bodyvisual, Bped1985, Brick Thrower, Brighterorange, Bryan Derksen, Btx40, BurntSky, CWenger, Can't sleep, clown will eat me, CanisRufus, Carlmewis, Celi0r, Ceycocky, Charles Matthews, CharlesC, Chris the speller, Christian List, Colonies Chris, Conversion script, Coop Rouge, Curvers, Cybercobra, DabMachine, Danakil, DerHexer, Dlacesz, DouglasR007, Dragentsheets, Dysprosia, Ed Poor, Eelis, Eelis.net, Esap, Fred Bradstadt, Furykef, Fuzheado, Gamer Eek, Gary King, GeorgeMoney, Ghewgill, GoingBatty, GoldenXuniversity, Gonzalo.vinas, GraemeL, Graham87, Hede2000, Hervegirod, Hibbleton, Hirzel, Hunting dog, Hyperfusion, IGLOBALIZE, Iancarter, Incubusurus, Ipcbung, J-Star, Jamesday, Japo, JB-adder, Jecar, Jeeves, Jerome Charles Potts, Jess, Jxalexand, John Vandenberg, Jondel, Joswig, Joycef, Jutiphan, K.Nevelsteen, KayEss, Kim Bruning, Knutux, Kurykh, Kwamikagami, Legare, Leszek Janiczuk, Libertyerie2, Liftarn, Ligulem, Linus M., Lit, MIT Trekkie, MMSequeira, MParaz, Maian, Mann jess, MartinHarper, Masklinn, Mathurnitn, MattGiua, MattOates, Metaheurist, Michael Hardy, Mild Bill Hiccup, Milo03, Minghong, Mintyng, Mistercupcake, Mma113, Msadegi, Naggarwal.nikhil, Narcelio, Nfvs, Nicelygedc, Ninjakannon, Njlover, Noophyte, Nsaa, Numbo3, OIEnglish, Oran, Orderud, Palfrey, Patrick, Pcap, Peyre, Phraine, Pion, Pmiliu, Pnm, Politepunk, Porcherceptor, Prari, Prashanthomesh, RaulMiller, RedWolf, Redreign, Remi0o, RexNL, Ronyclau, Rs kumar b4u@yahoo.co.in, Rursus, Rz, SJF, SP-KP, Sachingupta, Sae1962, Sahirshah, Salix alba, Sam Hocevar, Sdorman, Shawn wiki, Shoeossss, Simtay, Sippel2707, Smarmad, Solarra, Stavarot, Stephan Leeds, Stv, TakuyaMurata, Taw, TechTomy, Tech2ee, Teetaeweepo, TestPilot, Tgeairin, The Evil IP address, TheNightFly, Thecheesykid, TheJulienne, TimBentley, Time3000, Timneeu22, Tobias Bergemann, Torc2, Twocs, Unfree, Unixxx, Ver, Vera Cruz, VictorAnyakin, Wakimakirolls, Wernher, Whitepaw, Wlievens, Woohooikit, Xcvista, Yms, Zedlik, 424, זכ' נ' י' anonymous edits

Object (computer science) *Source:* <http://en.wikipedia.org/w/index.php?oldid=515595036> *Contributors:* 16@r, 3ICE, AGK, Acroterion, Ahsteele, Altenmann, AnAbsolutelyOriginalUsername42, Anwar saadat, Avaloan, Balabiot, Barauswald, Bevo, Blue2, Bookinvestor, Br77rino, Bryan Nguyen, Calorus, Cameltrader, CarlHewitt, Charlie.salts, Chealer, Chuunen Baka, Coffeehood, Curvers, Cybercobra, DabMachine, Danakil, DerHexer, Dlacesz, DouglasR007, Dragentsheets, Dysprosia, Ed Poor, Eelis, Eelis.net, Esap, Fred Bradstadt, Furykef, Fuzheado, Gary King, Grblomther, Green caterpillar, Harborsparrow, Hede2000, Helenem, Hervegirod, Horselover Frost, Icsearturles, InShanee, J991, Jacob Finn, JdkBac, Jerome Charles Potts, Jerryobject, JesseHogan, Jorge Stolfi, K0n2ad, Kdakin, Keepssouth, Kendrick Hang, Knutux, Krzys ostrowski, LiDaobing, Libuc, Lugia2453, Malleus Fatuorum, Martarius, Martpol, Mattbr, Mdd, Michael Hardy, MikeSchinkel, Mintyng, Mintyleaf, Musiphil, Narcissus, Nighthowl12, Noldoaran, Ottomachin, Patrick, Paul Foxworthy, Pcap, Pedant17, Phe, Phoe6, Pie4all88, Pinar, PlaysWithLife, Polluks, Prashanthomesh, Quilokos, R. S. Shaw, Rakwiki, Rbakels, Robbiemorrison, Ronz, Shivi.sidhu, Sigra, Snideology, Spoon!, Stdazi, Synaktion, Ta bu shi da yu, Taemyr, Taibah U, Taka, TakuyaMurata, The Berzerk Dragon, Thecheesykid, Tobias Bergemann, TonyClarke, Tooto, Ultimatewisdom, Utoloti, Vera Cruz, VictorAnyakin, Where, WikHead, Williamburg, Wlievens, Xxeo, Xqsd, Zabdiel, 168 anonymous edits

Object-oriented programming *Source:* <http://en.wikipedia.org/w/index.php?oldid=520773798> *Contributors:* *Kat*, ---Sol---, 0x7c2, 10metreh, 1exec1, 209.157.137.xxx, 2help, 4th-otaku, 925fahsfkh917fas, A More Perfect Onion, AGK, Abdull, Abhaybharat, Abhishek191288, AdamCox9, Adjusting, ADMn, Adu015, Ahmed505, Akashthukral, Alai, Alansohn, Algernon, Algorhyme, Alienus, Alismayilov, Allan McInnes, Allen Moore, Anastrophe, Ancheta Wis, Andante1980, Andre Engels, Andrew Eisenberg, Andropod, Andy Dingley, AngryBear, Anna Lincoln, Annalise, Anonymous Dissident, Antixt, Antonielly, Applepwln, Apurvugpta1996, Arthena, Ashmoo, Atas76, Atozjava, Autarch, AxelBoldt, Az29, Babij, Backslash Forwardslash, Bart133, Barte, Batslayer, Baudway, Bdodo1992, BeeJay, Beland, Beliavsky, Ben Standeven, BenFrantzDale, Bennie Noakes, Beno1000, Bernard van der Wees, Bevo, Bibikoff, Bigjmr, BinaryBlt, Biterid, Bjelleklang, Bkwllwm, Blehfu, Blouis79, Bluemoose, Bmejis, Bobo192, Bobzchemist, Bogdanmimousi, Boing! said Zebedee, Bongwarrior, Boy.pockets, Brent Gulanowski, Brick Thrower, Bucketsoft, Burger9191, BurntSky, Cabef6403, Calimo, Callmekosh, Cameltrader, Can't sleep, clown will eat me, CanisRufus, Capi crimm, CarlHewitt, Carrig, Catamorphism, CatherineMunro, Cburnett, Cdc, Cerberusrunning, Charles Matthews, Chatfector, Chcknwmm, Chealer, Chigger1111, Chowbok, Chris Roy, ChronoKinetic, Chuckhoffmarr, Clement Cherlin, ClementSeveillac, Clonts01, CloudNine, Cokoli, Colin Angus Mackay, Comperr, Conversion script, Cosman246, Countersubject, Craig Stuntz, Crazybilly, Crys, Cwolfsheep, D15L3X51CK, D3Jones, DBetty, DNewhall, Danakil, Daniakhtar, Darius Bacon, DarkStar, Darkwind, David gv ray, Dawid, Db24355, Dclf, DeadEyeArrow, Decltype, Dextakomm, Dgreen34, Disavian, Dominus, Donaldino, Doomer D, Great, Doopdroop, Doradar, Dpol, Dratman, Dschach, Duffman, Duja, Dustimagic, Dwheeler, Dysprosia, E, Ripley, EMaringolo, ESkog, Earear, Eclipse 89, ElTyrant, Ellyxc, Elwikpedista, EngineerScotty, Ebpr123, Equend1, Erget2005, Erjicul, Eritain, Esap, Expertus, Etrigan, Everyking, Evil saline, Ezani, FaerieInGrey, Falcon8765, Fbartolom, FellGleaming, Fennasnogothrim, Fennec, FinalMinuet, Finlay McWalter, Fireman biff, Fisherm77, Flashmob, Flemirna, FlinkBaum, FlyHigh, Fordmadoxfraud, Formankind, Fred Bradstadt, Fredrik, Freshbaked, Fuchsiyas, Funkelnagelneulin, Furykef, GRuban, Gadfin, Gaius Cornelius, Gareth Griffith-Jones, Ghassi, George The Dragon, Gerbrant, Geregen2, Giftlife, Gillwill, Glacialfox, Glen, Glenn, Golbez, GraemeL, Graham87, Graham, Greg trester, GregorB, Gscshoyu, Gwern, Hadal, Hairy Dude, Hajhouse, HamburgerRadio, Hamnes Hirzel, Happytailor, Harborsparrow, Hayabusa future, Hedgehog83, Henrygb, Hervegirod, Hirzel, Hlg, Hosick, Howenfink, Hu, Hu12, Hugo7m, Husond, I am neuron, IMSoP, Ianalis, Iluvcapra, Imjake9, Immunize, IndianGeneralist, Ionescuac, Ipson, Ivan Štambuk, J.delanoy, J36miles, JDP90, JForget, JLaTondre, JMSwtlk, JRR Trollkien, JaK81600, Jaalto, JadeInOz, Jalesh, Jarble, Jauerback, JavaGroupie, Javidjamae, Jaxelrod, Jaylewis, Jaysweet, Jayvelocity, Jdfote, Jeandré du Toit, Jeffreykegler, JenVan, Jnenavecia, Jeronimo, Jerryobject, Jftekna, Jhomme, Jim1138, Jimcreate, Jk2q3jrkls, Jleedev, Jncraton, John Vandenberg, John254, Joswig, Jpane1, Juhtolv, Jung dalgish, Jusdafax, K.lee, KJedi, KSeti, Kdakin, KellyCoinGuy, Kelpin, Kendrick Hang, Kenny sh, Kenorasis, Ketil, Kharevikas, Kharissa, Kilom691, Kivar2, Kku, Knutus, Komet, Krilon, Krisinbloush, Kukini, Kwamikagami, Kylu, Leandro, Lee Daniel Crocker, LeonardoGregorian, Lerdthenerd, Leszek Janiczuk, Letdorf, Lethaniol, Lethe, Lexspoon, Liao, Lightmouse, Ligulem, Liimes, LilHelpa, LinguistAtLarge, Loadmaster, Loekbergman, Lombana, Lone boatman, Lost goblin, Lotje, Luna Santin, M2Ys4U, M4gnum0n, MER-C, MIT Trekkie, Macarion, MagnaMopus, Mahanchian, Majilis, Malleus Fatuorum, Malthazar, Mankarse, Mann jess, Marcus Qwertus, Maria C Mosak, Mark Renier, MarkSpanier, MartinSpanier, Marudubshinki, Masiano, Materialscientist, Matt.zellman, Mattisgo, Matusz, Mauror, MaxHund, Maximus Rex, Mayankkapoor, Mcripper, Mdd, Mean as custard, Menacer, Mets501, Mia ib, Michael Hardy, Michael Clerx, MikeSchinkel, Miknight, Mikon, Mikrosam Akademija 7, Millahnna, Mindbuilder, Minesweeper, Minghong, Mintguy, Mintleaf, Mishagale, MisterSheik, Mithaca, Miym, Mjb, Mjchonoles, Mkarlesky, Mmernex, Moanzhu, MonoAV, Moonwick, Mordomo, Mountain, Mr snarf, MrChupon, MrFizyx, MrMambo, MrOllie, MrsGorilla, MsZger, Mtking, MuZenike, Music Sorter, Mxn, My another account, N998, NAHID, NHRHS2010, NW's Public Sock, Nagy, Nagytibi, Nakon, Nameneko, Nargs 2008, Nchaimov, Ncjones, Ndenson, Neilc, Nerdmonkey, Nick Number, Ninjatails, Ninly, Nitewinheils, Nitrambarrell, Nixdorf, Nnp, Nobody Ent, Norm mit, Normxxx, Ntse, NuclearWarfare, Nyttend, O.Koslowski, Obligato17, ObsidianOrder, Ocaasi, Olivier, Omicronpersei8, Omnipaediast, OnesimusUnbound, Only2sea, Orange Suede Sofa, Orderud, Otterfan, Ozmn, Ozten, P0lyglut, Plh3r1e3d13, P99am, PJTrail, PJonDevelopment, Pamri, Paradok, Paul August, Paul Magnusson, Paul Richter, Pavel Vozemilek, Pcap, Pcb21, Perrynewton, Peruvianllama, PeterC, Peyre, Pharaoh of the Wizards, Philip Trueman, Phoe6, Physis, Piano non troppo, Piet Delport, Pomoxis, Powtch, Prari, Pseudomonas, Puzzlesmith, Qwerty, RFClover, Rahzel, Raise exception, RaulMetumtam, Raven in Orbit, RedWolf, Rednblu, Remmus7, RevZoe, Rfc1394, Rich Farmbrough, Rileydutton, Riventre, Rjd0060, Rjwilmsi, Rob Grainger, Robby rocking, RobertMfromLI, Roger McCoy, Rrezov, Rsantmann, Rursus, Ruud Koot, SDC, SPUI, Sae1962, SalinasJ, Salix9ice,

Sam Hocevar, Samsara, Sandipsandilyasonu, Sango123, Schoelle, Scotjmcdermid, Scottishgary, Seaphoto, Seffer, Shaddim, Shadowjams, Shaji1997, Shamanx, Shamatt, Shayno, Shmilyshy, Shreevatsa, Sigmundg, Signalingz, Sir Link, Sjakkalle, Sjoerd visscher, Skizzik, Slakr, Slike, Smiler.se, Snaxe920, Solpeth418, SonGoku10, Spandrawn, Spicymickey, Splash, Spliffy, Sspecter, Statsone, Stephenchou0722, Stephenh, SteveSims, Stiang, Subdulous, Subtractive, Surturz, T00h00, THEN WHO WAS PHONE?, Ta bu shi da yu, Tablizer, Taemyr, TakuyaMurata, Tantrumizer, Tarcieri, Tarquin, Tassedethe, Taurrandir, Taw, Tawker, Tbhotch, TechControl, Tedmund, Teknic, Texture, That Guy, From That Show!, The Thing That Should Not Be, TheGrappier, ThePinkEmoRanger, Theopolisime, Thim_AntiquePen, ThomasO1989, ThomasOwens, ThomasUnivalor, ThreeVryl, Thucydides411, Tide rolls, Tigrou-Grenoble, Tim32, Tinalles, Tobias Bergemann, TonyClarke, Torc2, Tstormcandy, Tvarnoe, Tygrr, Ukepxat, Uncle G, UncleDoggie, UnicornTapestry, Ursula Huggenbichler, Uturseh, UttersRG, V R Kiran, V31a, VKokielov, Varuna, Varayil, Veledan, Vera Cruz, Violetriga, Vrenator, W.F. Siu, Waffle, Walt373, Warraqeen, Wavelength, Weylinp, Whispering, Widr, Wikinauta, Wikipelli, Wiresse, Wlievens, Wolfnx, Wtmitchell, Ww, Wyatt915, X42bn6, XDanielx, Xmanmonk, Xqsd, YakuShima, Yamamoto Ichiro, Yath, Yekrats, Yellowdesk, YemeniteCamel, Ykhwong, Yminsky, Zabdiel, Zachlipton, Zayani, Zeno Gantner, Zoeb, Zondor, Zron, Zven, Åvar Arnfjörð Bjarmason, Σ, 1683 anonymous edits

Smalltalk Source: <http://en.wikipedia.org/w/index.php?oldid=511314197> Contributors: 130.94.122.xxx, 16@r, 18.94, 2ndMouse, 96.186, Acacix, Adrian two, AlistairMcMillan, AllanBz, Alphajuliet, Amnariix, Andre Engels, Andrew walker, Andyzweb, Antonielly, Apauley, Avakar, Balabiot, Bartleby, Bbartlog, BelAus, Ben-Zin, Bevo, Bleakgadfly, Blouis79, Blueshifter, Brianegge, Bronzemirror, Bryan Derksen, Bshea, Canaima, CanisRufus, CapitalSasha, CarlHewitt, Chinju, Chocolately, Chowbok, Chris-gore, Chuunun Baka, Cmdrjameson, Codepoet, CodingBucky, Coffee2theorems, Conversion script, Cybercobra, DNewhall, Dachshund, Danakil, Dancewiththesky, David Woodward, Dcoetze, Dekails, Demonkoryu, Derbeth, Dere Ross, DeweyQ, Diana Merry-Shapiro, DisasterManX, Disavian, Dmsar, Drj, Dsmdglod, Dysprosia, EMaringolo, EagleOne, Edward, ElAmericano, ELBenevolente, Eliotmiranda, Emplynx, EngineerScotty, EricEnfermero, Erik Corry, FatalError, Fatespeaks, FormerNukeSubmariner, Fredrik, Furykef, Garygregory, Genifycom, GeorgeMoney, Gokusandwich, Golubovsky, Graham87, Grendline, Grshiplett, GruberGuy, Guidelove, Gwern, Hairy Dude, Hannes Hirzel, Helicon, HereToHelp, Hinrik, Hirzel, Hlegius, Hooperbloob, Hraefen, Hroðulf, Husond, Ire and curses, Itai, JLaTondre, Jakobou, Jacj, Jagun, JasonSayers, Javy tahu, JediKnigghte, Jerome Charles Potts, Jerryobject, Jfizzell, Jimmyzimms, Jithinkuiva, Jleedev, John Nevard, Jordandann, Jrthorpe, KKong, Kaicarver, Keyanoo, Kpjais, Kricxjo, Kylealanhe, LVC, Ladder to Heaven, Laug, Lerdwsua, Looxix, Luvercraft, Madher88, Mal4mac, ManuelGR, Mariya Oktyabrskaya, MattGiua, Matthewdunsdon, Mattoothman, Mboszko, McGeddon, Mcaruso, Mcmccann, Mcsee, Mdjohns5, Meddin, Metageek, Mf08cygb, Mggreenbe, Michael Hardy, Mipadi, Mivsek, Modulatum, Mortense, Mountain, Mr_Shoelss, MrSteve, Msiddalingaiah, Mulder416, Nanshu, Njones, Nemo2000, Neurocod, Nhunter, Nicguyedc, Nickj, Nixdorf, Nozebacle, Octahedron80, OfficeGirl, Ojophoyimbo, Oliver Pereira, Omerch2, Oneiros, Pablomarx, Paul Richter, Paloslump, Pbukra, Pdveloman, Pgr94, Phil Boswell, Piano non troppo, Pmcjones, Poor Yorick, Quuxpluseone, Qwertys, R4p70r, Rageausaurs, Rajahk, RandalSchwartz, Rangi42, Ranpháirtí anaithnid, RedWolf, Renku, Rockear, Ruud Koot, SL93, Salag, Sam Hocevar, SchreiberBike, Sinewalker, Slady, Slocombe, Smallscript, Sole Soul, Sqrmx, StaticGull, Stefan Urbanek, Strake, Stumps, Subsolar, Sukiari, Sysy, THEN WHO WAS PHONE?, Taam, Taw, Template namespace initialisation script, Tesler, Thomasmalen, TimBray, Tkavle, Tobias Bergemann, Tompsc, Torc2, TotoBaggins, Trojan2k, Ubergeek3141, UttersRG, VX, Vassili Bykov, VassiliBykov, Vidkun, Wapcaplet, Wavelength, WendelScardua, Who, Wickorama, Wikilolo, Willsutt, Windharp, Wlievens, WojPob, Xx1foren, Yath, Ytouquet, Yworo, Zackzack, Zoiicon5, 442 anonymous edits

Common Lisp Source: <http://en.wikipedia.org/w/index.php?oldid=513584092> Contributors: -Barry-, Adriatikus, Ahoerstemeier, Amwyll Rwden, Anapazapa, AnonMoos, Aputtu, Arjun G. Menon, ArnoldReinhold, Aurelien, BenKovitz, BiT, BillAlexander, BlakeStone, Bleibach, Bowman, Burschik, Cadr, Charles Matthews, Cmcormick8, Coder Dan, Coffee2theorems, Coneslayer, Conversion script, Curly Turkey, Cybercobra, Danakil, DanielWeinreb, Danim, Dave Foley, David Gerard, Davidpk212, Dbmikus, Dcoetze, DeXXus, Demitsu, DjOnes, Dougher, Dront, Ds13, Dysprosia, E40, Edward, Erights, Estevoaei, Fred Birchmore, Fred Bradstadt, Fredokun, Fredrik, Fubar Obfusc, Furykef, G20071221, Gadflum, GermanX, Gesslein, GregorB, Gronky, Grue, Gudeldorf, Gwern, Imran, Inkdedm, Ircicq, Isaac Dupree, Isnow, Ivan Štambuk, J.H. McAleely, JMSWlk, James Crippen, Jcbcaudo, JeffZ, Jerryobject, Jevansen, Jmreinhart, Jocap, Jorge Stolfi, Joswig, Jside, Kaobear, Karolrvn, KazKylheku, Kenyon, KimDabelsteinPetersen, Klodolph, Kraken, KublaChaos, L33tminion, Lambda-mon key, Lbs6380, Leolaursen, Levin, Lkestelot, LodeRunner, Logan, MC10, Makrakis, MadEwkHerd, Madmardigan35, MagV, Magiluke, MarSch, MartinCracauer, MartyMcGowan, Marudubshinki, Matthewdunsdon, Mboverload, Micropolygon, Mild Bill Hiccup, Mindmatrix, Minesweeper, MoraSique, Mwtwoeis, N5ilh, Neilbeach, Neilc, Netizen, Netsettler, Nick Levine, POlyglut, PDG, Peotrovitch, Pfunk42, Phe, Pissant, Plugwash, Populus, Prmai, Qbg, Quasaur, Qwasty, Rabillperson, Rais exception, Ram-Man, Raw.com, Rbonval, Renku, Retired username, Rich Farmbrough, Richard coeks, Richardmstallman, Ruud Koot, SF007, Saccade, SamuelTheGhost, Sanxiyn, Sbroy88, Schissel, Seth.ryan, Sfemigier, Shlomif, Sibian, Siroxo, Smartse, Someone65, Somerct, Soulman, Starnestommy, Stassats, Stesch, Sverdrup, Tacitus Prime, Tackat, Tarka, Tassedethe, Template namespace initialisation script, Terjen, The Photon, Theowlow, Thoughtactivist, Thumperward, Thüringer, Tobias Bergemann, Tony Sidaway, Trevor Spiteri, Ursafoot, UserGoogol, UttersRG, VeryVerily, VictorAnyakin, Wavelength, Who, Wickorama, Wikieditoroftoday, Wingman417, WojPob, Yworo, Zamb, Zsam, Σ, 256 anonymous edits

OCaml Source: <http://en.wikipedia.org/w/index.php?oldid=520717345> Contributors: AaaghItsMrHell, Academician, Adrianwn, Aij, AlanBarrett, Alsandro, Andy Dingley, Argav, Arjun G. Menon, ArlenCuss, Austin Hair, Ben Atkin, Blue Prawn, Bobak, Bogdanb, Brianherman, Brighterorange, BurntSky, CBM, CRGreathouse, Cadr, CanisRufus, Cdammama, Cdunn2001, Cek, Cetincept, Cfcompte, ChriS, Chris-gore, Clayrat, Clonway, Clementi, Coder Dan, Cragwolf, Cybercobra, DNewhall, Danakil, DanielKO, Darioteixeira, Darklilac, David-Sarah Hopwood, David.Monniaux, David.deharbe, Dbaron, Dbelange, Dcoetze, Defective, DenisDiderot, DmitDiderot, Donbalcon, Donperk, Doradus, Dysprosia, Eloquent, Erik Sandberg, ErikHaugen, Error, Family Guy Guy, Fdrepas, Femto, Ferengi, Flamingspinach, Forage, Freidom, Furykef, Fuzzy00, FvdP, Gesslein, Gioto, Gmarceau, Greenrd, Gregorlowski, Gwern, Haakon Haeleth, Hari, Harvester, Hdante, Homerjay, Igordad, Igouy, Isaacedealy, Isnow, Itsmesid, JLaTondre, Janto, Jdh30, Jeff3000, Jerryobject, Jonas August, Jrouquie, Jrthorpe, Jsnow, Junkblocker, JustinWick, Jvs.cz, K.lee, Koper, Kuszi, Kwamiakami, Latitudinarian, Leandro, Leonidas from XIV, LittleDan, Logixoul, Macbutch, Madchester, Magnus Manske, Mapfn, MarkSweep, Martin Jambon, Martin.jambon, Marudubshinki, MatthiasWander, Max, Maximaxim, Mbac, Mbh, Mblumber, Mehdi311ggg, Mellum, Mhalcrow, Michael Hardy, Michalis Famelis, Miciyah, Mike Lin, Mindmatrix, Minesweeper, Mk*, Msnicki, Mww, Neilc, Nethgirb, Nick Nolan, Nimdil, Niteowlneils, Nmmogueira, Notepad, Nysin, Oconnor663, Ojigiri, Orphu of io, Peak, Pengo, PeterBrett, Pne, Populus, Positron, QrczakMK, Quackor, Quasipalm, Qwertys, Raboof, RedWolf, Requestion, Richard W.M. Jones, Ruud Koot, SF007, Sac1962, Scott Paeth, SeanProctor, Seriouslyuguis, Shlomif, Smimrom, Somerct, Spoon!, Stassats, SteinbDJ, SteloKim, StevenJ, Sunny256, Svick, Tagus, TakuuyaMurata, Taurimmas, Taw, Template namespace initialisation script, TheVoid, Theone256, Tobias Bergemann, Tompsc, ToohrVyk, Torc2, TuukkaH, Tyler-willard, VX, Vegaswikian, VictorAnyakin, Vincent Hugot, WayneMokane, Wernher, Wickorama, Wtachi, XPEHOPE3, Yahoolian, Yminsky, Yoric, Zaczhiro, Zanypoethboy45, Zeno Gantner, Пика Пика, 254 anonymous edits

Prototype-based programming Source: <http://en.wikipedia.org/w/index.php?oldid=509794135> Contributors: 2988, 4th-otaku, A-Day, A5, Aftereight, AlcoholVat, Andrew Eisenberg, Andrej Balaguta, Antonielly, Apankrat, Avinashm, B0an, Badanedwa, Bleakgadfly, Booty443, Brian.L.ODell, Bryan Derksen, Burschik, CRGreathouse, Chaos023, Cwolfsheep, Cybercobra, Danakil, DanielAmelang, Davigoli, Dbabbitt, Dcoetze, Diego Moya, Dougher, Ebraminio, Eekster, Espur, Eurleif, FatalError, Ffangs, Flammifer, Fredrik, GGShinobi, Gcanyon, Gerrit, Giardante, Golbez, Graham87, GreatWhiteNortherner, Gurklurk, Harlan879, Iggywangi, Indi, Jarble, Jason.grossman, Jerryobject, Jesterxl, JimClark, JoaquinFerrero, Jon Awbrey, Jpvinnal, Karvendhan, Kgaughan, Klondike, Krenair, Leaflord, Leonard G., Lerduw, Lingwitt, LittleDan, LittleSmall, Matthewdunsdon, Maury Markowitz, Marcika, Marcoxa, MattGiua, Maury Markowitz, Miguel.v, Miquonranger03, Mmick66, Modulatum, Mr2001, Msnicki, Nasnema, Netytan, Nice Stobie, Nihilires, Octahedron80, Oleg Alexandrov, Only2sea, Ontologiae, PatriceNeff, Pgan002, Phil Boswell, Phorgan1, PleaseStand, Polbrian, QuickBrownFox, RG2, Rabidphage, Ranpháirtí anaithnid, Rich Farmbrough, Ryandaum, Sagaciousuk, Sam Hocevar, Spp, Shen zq, Shemme, SirShanus, SlippyD, Stevage, Stevedekorte, Tabledhote, Tagae, That Guy, From That Show!, The Thing That Should Not Be, Thewinch, Tobias Bergemann, Toussaint, Transobj, Valafar, Vdm, Vicoar, Wlievens, Zzedar, 160 anonymous edits

Self (programming language) Source: <http://en.wikipedia.org/w/index.php?oldid=515851761> Contributors: -- April, Ark, Badanedwa, Bejnar, BelAus, BillFlis, BirgitteSB, Brion VIBBER, Buckwad, Charles Gaudette, Cjpuffin, Cybercobra, DNewhall, Danakil, Dd75sg, Derek Ross, Dougher, Dysprosia, Ellmist, EnTerr, EngineerScotty, Erik Garrison, Ffangs, Floorsheim, Fredrik, Fuhghettaboutif, Furykef, Gerwerk, Ghilly, GneralTsao, Golbez, Graham87, Headbomb, Hossein bardareh, House, Jamelan, Jarble, Jecel, Jerryobject, John Nowak, K.lee, Karl-Henner, Kgoarany, Ladislis Mecir, LiDaobing, Lingwitt, Linuxbabu, LittleDan, LittleSmall, Matthewdunsdon, Maury Markowitz, Mcsee, Michael Hardy, MisterSheik, Mountain, Mn, Neilc, Noldoaran, OmerMor, Ontologiae, Paddu, Pavel Vozenilek, Pdemb, Phil Boswell, RandalSchwartz, Rememberway, Rotty, Ruakh, Rudimae, Serverside6, ShaeErsson, StarFisher, SilverStar, Stw, Talandor, TaleTreadar, Tarquin, Teapeat, The Anome, Tobias Bergemann, Toddasmith, TonyClarke, Trey314159, Twang, Ultrasaurus, VX, Valeriya, Vdm, Waggers, Wavelength, Wik, Wolfkeeper, Yworo, Zoiicon5, 83 anonymous edits

JavaScript Source: <http://en.wikipedia.org/w/index.php?oldid=520774525> Contributors: 12 Noon, 16@r, 1wolfblake, 31stCenturyMatt, 80N, A bit iffy, ABCD, ABF, ANGGUN, Aapo Laitinen, Aaroniba, Abc123456person, Abhishekitmmb, Abhkum, Acostin, Adam Hauner, AdamTRieneke, AdmN, Aeons, Afrobuddy, AgentCDE, Ahda, Ahoerstemeier, Ahy1, Ajfweb, Aka042, Akronym, Alansohn, AlastairIrvine, Alisha.4m, AlistairMcMillan, Allens, Aman2008, Amgc56, Amire80, AmitDeshwar, Ancheta Wis, Andre Engels, Andreas S., Andrew Levine, AndrewHowse, Andrewrost3241981, Andy Dingley, AndyZ, Andytuba, Anferryjohnsons, Ankles, Anna512, AnnaFrance, Antimatter15, Antipastor, Antonio Lopez, Aoi, Apollo1758, AquaGeneral, Ardonik, Arjayay, Artemis-Arethusa, Artw, Arvindn, Asqueella, Astrobloby, Avijitguharoy, Avoided, AxelBoldt, B0at, B1KWikis, BBL5660, BBilge, BP7865, Bacon and the Sandwich, Badman11, Bamkin, Barek, Beau, Belgar, Ben-Zin, BenAveling, Bencherlite, Bender235, Benjaminikoakes, Bergie, BernhardBauer, Betax, Bevo, BinaryWeapon, Bionicsraph, BirdieGalyan, Blobglob, Blonkun, Bob Re-born, Bobcat hokie, Bobdc, Bogtha, Boing! said Zebedee, Bonadea, Boomshadow, Boshom, Bpeps, Brainyiscool, Bratsche, Brendan Eich, Brettz9, Brian Kendig, Brighterorange, Brilliant trees, Brion VIBBER, Bryan Derksen, Butch566, Bwicker, CFOrrester, CO, CQ1, Caiguanhao, Caltas, Calvin 1998, CambridgeBayWeather, CanadianLinuxUser, CanisRufus, Canonical Rush, Cap'n Refsmmat, Capricorn42, Carewolf, Carey Evans, CaseyPenk, Cash cab, Cat-five, CecilWard, Cenarium, Centrx, Chairboy, Chaojoker, Charles Gaudette, Charles Ilya Krempaux, Charles.2345, CharlotteWebb, Chazwatson, Chealer, Chester br, Choronzon111, Chris the speller, Chrisdolan, Chrisman247, Christian75, Chriswiki, Chuq, CiudadanoGlobal, Civilis, Cliciu, Closedmouth, Cldsennis2007, Cmdrjameson, Coffee, ColinHelvensteijn, ColinJF, Collabi, Combustablejo, Corti, Cpu111, Crazycomputers, CreamPies, Cronium, Crystallina, Cst17, Csusbdt, Cwilm2, Cybercobra, Cyfal, D.brodale, DARTH SIDIOUS 2, D0Neil, DTR, DVdm, Damian Yerrick, Damicatz, Danakil, Daniel.Cardenas, Daniel5Ko, Dantheman531, Daonguyen95, Darkride, Dasch, DaveK@BTC, Daveed84x, Davyedweeb, David Edgar, David Gerard, David Wahler, David1217, DavidCary, Davidiad, Davigoli, Dbabbitt, Dcoetze, Delcnsltmd, Delfuego, Demonkoru, Den fjärtade ankän, DennisWithem, Deor, Derekleungsheets, Desliblast, Dhml, Dhmtlkneches, Diannaa, Digita, Discospinster, Dlexc, Dlrohrer2003, DocWatson42, Dodecagon12, Don4of4, DonToto, Donber, Dragon Dave, Dreadstar, DreamGuy, Drizzt, Drsqurzl, Dyedarg, Dysprosia, Eagleal, Echo95, Ed g2s, EdC, EdJohnston, Edfrommars, Edgar181, Edknl, Edward Z. Yang, Eksplori, Elefectoborde, Eliashedberg, Eliot1785, Ellmist, Emperorborg, Engelec, Enkauston, Enmajm, Ensign beedrill, Epochbb, Er Komandante, Erich gasboy, Erkcan, Eskimospy, EvanProdromou, Eventualentropy, Everyking, Ex-dude329.4, Excirial, Execvator, Explicit, Extremecircuitz, Face, Faisal.akeel,

Familyguy0108, FatalError, Echoong, Felipe1982, Femmina, Fetchcomms, Fistboy, Flager, Florencefc4eva, FootholdTechnology, Fraggie81, Frap, Frau Holle, Frecklefoot, Fred Bradstadt, Fred Gandt, Fredrik, Func, Furykef, Fwerdna, GTBacchus, Gail, Galwhaa, Gamaliel, Garycl, Gary King, Gaurav21r, Geary, Geneb1955, Genhuan, Geoffrey, Georgryp, Gerbrant, Gerweck, Ghettoblaster, Giftlite, GigAtomixX, Glass Sword, Glen 3B, Gm4, Gogo Dodo, Goplat, Gorpik, Graeme Bartlett, Graemel, Graham87, Grayczyk, Greenie2600, Greenminz, Gregersrygg, GregorB, Griba2010, Grika, Google, GroovySandwich, Grunny, Guaka, Guitardemon666, Gunug, Gurchzilla, Gwernol, Gzabers, HDRAKE, Hairy Dude, Hannes Hirzel, HappySailor, Hbckman, Hblank, Herover, Herveigrod, Hex, Hgmichna, Hirzel, Hmainas, Hongguo, Hoo man, Hosmich, Hu, Hu12, HundUno, Hydrargyrum, HyperCapitalist, I Mystic I, I already forgot, I3enhamin, IRP, Iamjapatel, Iamyourdome61, Ian Bailey, Ida Shaw, Idiotfromia, Ieee8023, Igoldste, Illoeobnbbu, Illicium, IlliterateSage, ImperatorExercitus, Inseisyou, Insouciance, Integr, InvertRect, Iridescent, Irish Souffle, Irrbloss, Itpastorn, ItsZippy, IvanLanin, Jdelanoy, JFG, JForget, JVz, Jack Waugh, JackyBrown, Jacobolus, Jagginess, Jake Nelson, Jake Wartenberg, James.abley, JanInad, Jarble, Jasper Deng, JavaKid, Javalikescript, Jay, Jec, Jeresig, Jerryobject, Jesant13, Jfd34, Jim McKeith, Jim1138, Jlerner, Jmartinsson, Jnigam33, Jobanjohn, Jodia.schneider, Joeyte50, Joffeloff, JohnCD, JokarMan, JonathanAquin0, Joncnum, Jonhanson, Jor, Jpgordon, Jtleitz602, JuanIgnacioGlesias, Juliancolton, Jumbuck, Justie1220, JustinStolle, Jwadeo, Kenny TM-, Kenyon, Kesac, KeybladeSephi, Khaled0147, Kiensvay, Killab14, Kincc, King of Hearts, Kingpin13, Kinu, Klildiplomus, KnowledgeOfSelf, Knutkj, Koavf, Kozuch, Krenair, Krinkle, Kroum, Kshtiz005, Kungming2, L Kensington, Lacqui, Laesod, Lambyte, Lanxiazhi, Largoplatz, Lateq, LegendGamer, Ledgerbo, LeeU, Leg614, Lemon octopus, Leothill, Lesgles, Lethe, Lexprfunccall, Lhtown, Lianmei, Linuxerist, Luijiang, Looie496, Lucy75, Lukasblakk, Lukestanley, Lupin, MER-C, MFNICKSTER, MIT Trekkie, MVelliste, Mabdul, Macaldo, Machine Elf 1735, Madhusudan N, Magioladitis, Magister Mathematicae, Mahewa, Maian, Mangst, MarK, Marc-André Åbrock, Marcoxa, MarekPro, Mariano, viola, Mark Renier, Marklbarrett, Martarius, Martin.kopta, MartinRinehart, Martinheit, MatheoDJ, Matifibrithum, MattjaP, MattForestpath, Matthew Yeager, Maury Markowitz, Maximus Rex, Mazin07, Mbarone911, McDutchie, McSly, Mcclurmc, McKoss, Message From Xenu, Mfc, MichaelBillington, Microgoal, Miernik, Mifter, Mikeblas, Mikehtw, Mikenole, Mikething, Milefool, MilesAgain, Milly, Mindmatrix, Minesweeper, Minghong, Miranda, Misericord, Mistress Selina Kyle, MithunRT, Mjb, Mkweise, Mmick66, Mnemeson, Mnemonicof, Mob590, ModDJesus, Moeron, Mohammad khallad, Monority, Moonyfruit, Morgansutherland, Mortense, Mouchoir le Souris, Mr. Wheely Guy, Mr.Clown, MrOllie, Mrt3366, Mrwiggles, Ms2ger, Muntfish, Muzzamo, Mxn, MyPallmall, Mütze, NYKevin, Nako16, NamesIsRon, Nanshu, Nanud, Nashley, Nate Silva, Nate879, Nathan2035, Nczempin, Nealcardwell, Nearfar, Necromantarian, Neilc, Neilerdwien, Neilshermer, NeonMerlin, Neonario, NerdyScienceDude, Nezzite, Nicetygedec, Nick8325, NickW557, Nikfy, Nigeli, Nivix, Nixeagle, Njuton, Nu, Nlu, No Guru, Nol888, OLENJONNE123, Obiwankenobi, Odaravlaac, Official Spokesman, Onnoitsjamie, Oleg Alexandrov, Oli Filth, Oliphant, Oliver Pereira, Oneiros, Oobug, Opticyclic, OrangeDog, Orderud, Oros0, Osoviejo, Oxymoron83, OzFred, Pacoup, Paercebaf, Paradox7798, Patrick, Patrick Corcoran, Paul.irish, Paul1337, Peak, Pearl's sun, Peter L, Peter S., Phantombantam, Pharaoh of the Wizards, PhilHibbs, Philcha, PhilipR, Piano non troppo, Piet Delpot, Piotrus, Plest, PlusMinus, PointedEars, Poor Yoric, Prakshal, Presto8, Prince-of-Life, Prolog, Psalmstrist cheifprosperity, Psychtf, Psz, Pyrospirit, Qatter, Quale, Quamarret, Quantum00, QueenCake, Quinxorin, Quitezuce, R'n'B, R3m0t, RG, RJaguar3, Radagast83, Ramesh Chandra, RandomStringOfCharacters, Razorf, Rbirkby, Rbucci, Rdsmith4, Recnigilare, Recognitione, RedRollerskate, RedWolf, Reinalth, Reisio, RexNL, Rfl, Rgoodermote, Rrgld, Rhobite, Rich Farmbrough, Rich Janis, Rick Block, Rickyails, Ringbang, Rizome, Rjwilmsi, Robbie098, Roberto Cruz, Robmv, Rohitjs, Ronhjones, Rostz, Rp, Rrror, Rschen7754, Rsrikanth05, Rufous, Rufustfirefly, Rurus, Ruud Koot, Rwalker, Ryan Postlethwaite, S-n-ushakov, SC.M.Samee, SFC9394, SJP, ST47, Saken kun, Salarmehr, Sam Hocevar, Sam Korn, Samfosterian, Samliew, Samuell, Samwr1234, Sander Säde, Sango123, Saqib, Sayer5512, Sboy365, Scal.Motor, Scarpy, Sceeeup, Scuirinae, Selck23, Sealican, Seav, Selkel, Shalom Yechiel, Shanel, Shantirao, sharkD, Shenme, Shriram, Shuiqvz3, Shwaza, Simoes, SineQ, Singhivender, Sir Lewk, Sfjormen Skizzik, SkyWalker, Slamb, Sleepyhead81, Slevenitis, Slippyd, SlowByte, Smyth, Snowolf, Snoyes, SomeStrange, Someguyonthestreet, Spalding, SparrowsWing, Spe88, SpeedyGonsales, Spellmaster, Spiel, Spitfire, SpuriousQ, StaticGull, SteveSims, Steveholt, Steven Walling, Stickguy, StudioFortress, Stuffandthings, Sundström, Super Rad!, Superm401, Superslacker87, Surfeited, Susan Davis, SusanLesch, Sushiflinger, TFOWWR, TOReilly, Tabledhote, Tabrez, TakuyaMurata, TarnoK, Tasc, Taw, Techname, Tedickey, Template namespace initialisation script, Tgeain, Thatguylint, The MoUsY spell-checker, The Nut, The Parting Glass, The Rambling Man, The Thing That Should Not Be, The Wild Falcon, The wub, TheColdestFusion, TheKMan, TheMuffinWalkers, TheTechFan, Thefuzzballster, Thekaleb, Themasterned, TheOne256, Thing, Think outside the box, ThirdSide, Thron7, Thumperward, Tigerwol7753, Timwi, Tizio, To Fight a Vandal, Tobias Bergemann, Todd Vierling, Todcs, Toke, Tom Jenkins, Tophu, Toussaint, Toyotabedzrock, Tpk5010, Trigger hurt, Troels Arvin, Troy.hester, Trustle, Tsemii, Tunnable, Tyomitch, Typhoonhurricane, Tysto, UU, Udittmer, UltramanDK, Uncle G, Urbanus Secundus, Useight, Userask, Utcursch, VKokielov, Vasily Faronov, Velella, Versageek, Vertium, Violettriga, Vishwanatharendekar, Vocaro, VoluntarySlave, Vossfeld, Vrilupol, WoOzySh00t3r, Wadsworth, Waldir, Waldred, Wapecraft, Wasell, Waterfalls12, Wavelength, Wayward, Website-andrew, Weirdo134, Welithy, Werner, Who, Why Not A Duck, Wickorama, WikiLauren, Wikibob, Wikieat, Wikieditoroftoday, Winterst, Wisdom89, Wzwz, XP1, Xedret, Xhienne, Xudifsd, Yamakiri, Yamakiri on Firefox, Yaron K., Yellowdesk, Ylandra, Ylbissop, Ysangkok, Yuki Konno, Yuval Madar, Zaak, Zakawer, ZakuSage, Zhou Yu, Zigger, Zin, ZmiLa, Zoicon5, Zojj, Zootm, Zundark, Zven, Zvn, Zzuuzz, Σ, ຂອບ ເພີ້ມເພີ້ມ, 1657 anonymous edits

Metaprogramming Source: <http://en.wikipedia.org/w/index.php?oldid=517386031> Contributors: 5gon12eder, Adamarthuryan, Alansohn, AlistairMcMillan, Anders Kaseorg, Angusmclellan, Balabiot, Brick Thrower, Burschik, Candace Gillhooley, Capi crimm, Ceefour, Cramforce, Cybercobra, David Lamb, Dcoetzee, Derek Ross, Dpbsmith, Dratman, Ds13, Dysprosia, Edward, Egg, Elcabri, Emurphy42, Fansipans, FatalError, Feraudhy, Freshaconci, Funandtrvl, Furykef, Geh, Ghettoblaster, Greendr, Grelglewin, Grm wnr, Gro-Tsen, Grshtplett, Gwern, Hallabda, IanOsgood, Idean, JC Chu, JLATondre, Jason Quinn, Jaxelrod, Jerryobjet, Jonathan Grynpas, Josh Cherry, Kalan, Kesac, Kim Bruning, LinguistAtLarge, MD87, MDE, Manuscript, MarSch, Martyluo, Marudubshinki, Maxim.mazin, Mgsloan, Mpadi, Mlemos, Motor, Mstuoem, Mwid, Neile, Noldoaran, Northamerica1000, Obscuranym, Octahedron80, Pablo X, Perlyn06, Pgan002, Quuxplusone, R'n'B, Renku, Rheun, Rofthorax, Root4(one), Ryguas, Sae1962, Secretlondon, SethTisue, Shirik, Simetrical, Somnambulon, Strongssauce, T.vanschaik, TheAMmollusc, Theamazingusername, Thumperward, Timc, Toussaint, Treymeister, Vitaly Lugovsky, Xiaogaozi, Yb2, Yukoba, Zeroflag, 115 anonymous edits

Generic programming Source: <http://en.wikipedia.org/w/index.php?oldid=519567858> Contributors: 1ForTheMoney, Iexec1, A876, Abeliavsky, AlanUS, Albamhandae, Aldie, Ali Esfandiari, Angusmclellan, Antonielly, Ap, Apotheon, Art LaPella, Beliavsky, BenFrantzDale, Bezenek, Bluenoose, Brianbjparkar, Bryan Derksen, BurntSky, CanisRufus, Carlosayam, Cassowary, Cbogart2, Chininazu12, Chip Zero, Chl, Chris the speller, Cic, Cjoev, Clément Pillias, Creativename, CyborgTosser, DanielCarlsson, Daverocks, Dectype, Decrease789, Denispir, Dmeranda, Dm1Trix, Donhalcon, Doug Bell, Dreixel, Dungodung, Dysprosia, Ed Poor, Elwikipedia, Enerjazer, EuGuy, Epbr123, Eriksalt, Fastly, Fredrik, Furykef, GDonato, Gaius Cornelius, Gidonb, Gildos, Good Olfactory, Gurmeet, Hans Bauer, Hans Dunkelberg, HaoZian, Harald Hansen, Harborsparrow, Hasan aljudy, Heelmljnlevenlang, HenryLi, Hillel, Hsjawanda, IamOz, Immunion, Ionathan, Ivan rome, Jes5199, Jjgrainger, John Vandenberg, JohnOwens, Johnfos, Jorend, Josh Liu, Karada, Kejia, Kinema, KirkMcDonald, Krallja, Krischik, Lambiam, Leledumbo, Leo141, Lightmouse, LinguistAtLarge, Logan, Ludovic Brenta, Luiscantero, M4gnun0m, Maghnus, Marc Mongenet, Marudubshinki, MattGiua, Mathew0028, Mensch0815, MichaelPloujnikov, Mikademus, Mike Dillon, Mike92591, Miqademus, MisterSheik, Mistercupcake, Mitch Ames, Mmarshall, Mordomo, Mud7x7, Mysekurity, Nixdorf, NotASpammer, Nuno Tavares, Olutes, Onebit, Opelio, Orderud, Paddy3118, Patrik, Patriotic dissent, Pcap, Peter.alexander.au, Phil Boswell, Phresnel, Pomoxis, RedWolf, Renku, Robg, Ruud Koot, SeeSchloss, Shawn12341234, Shirik, Sjrodeaene, Skatche, Slaniel, SlowMovingTarget, Snaxe920, Softest123, Some jerk on the Internet, Soumyasch, SparsityProblem, Spoon!, Steevm, Sun Creator, Svick, T4bits, TRosenbaum, TakuyaMurata, Talandor, Thenickduke, Tim Starling, Tobias Bergemann, Torc2, Troels Arvin, Unnikrishnan.am, Vaughan, WRK, Wavelength, Whitepaw, Wile E, Heresiarch, Woohooikit, Xlcheng, YotaXP, Yunshui, Zian, 雲木諒二, 219 anonymous edits

Ada (programming language) Source: <http://en.wikipedia.org/w/index.php?oldid=519469594> Contributors: -Barry-, A. B., Abigail-II, Agent86, Ahoerstemeier, Avosto, Alan Peakall, Alan.pointdexter, AlbertCahalan, Alerante, Alexandru.chircu, Allen Moore, Andre Engels, Andrea Parri, Andrewpmk, Ap, Apokrif, Aranel, Archolman, Atanamir, Awg1010, Bawolff, Bemoeial, BenFrantzDale, Bevo, Bhfoland, Blorg, Bobby D. Bryant, Bobcalco, Bobde, Bobet, Borgx, Borislav, Boshomi, Brian Gunderson, Brianga, Brion VIBBER, BurntSky, C++ Template, CRGreathouse, Cander0000, Cgs, Chbars, Chowbok, Christian List, ClintGoss, Clown, Cogiat, ColBatGuano, Conversion script, Corti, Curps, Cybercobra, DNewhall, Danakil, Danrah, David.Monniaux, Dbmoodb, Dcoetze, Deb, Dectype, Demonkoryu, Denisaron, Denny, Diberti, Didier Willame, DonPMitchell, Dwheeler, Dysprosia, EgoWumpus, Elophas, Ems2, Epitome83, Esj, Evile, Evil saltine, Eyerland, Fanthrillers, Folajimi, Frecklefoot, Friendly person, FuelWagon, Fæ, Firat KÜCÜK, Gareth Griffith-Jones, General Wesc, Geniac, Georg Peter, Geregen2, Gerry Ashton, Graham87, GrandPoohBah, Grandscrubs, Gudelar, Guerby, Gzorneplatz, Hairy Dude, Hans Bauer, Hashar, Hcobb, Heartinpiece, Herove, Holizz, Hooperbloo, Horow021, Hughey, Iccaldwell, Intgr, J-p krelli, JLATondre, JWiliamCupp, Javaman59, Jcreem, Jcupak, Jeltz, Jerome Charles Potts, Jerryobject, Jhbdel, Jkominiek, Jnc, Johayek, Joonasl, Jorgenev, Jherutum, Kameo76890, Kel-nage, Klinealz, Knippi, Koyaanis Qatsi, Krischik, Lambert Meertens, LaurentMeilleur, Lightmouse, Ligulem, Lmat, Lostdj2, Ludovic Brenta, Luk, Lupinoid, Mac1958, Macrakis, Madbehemoth, ManuelGR, Marcushan, Mat-C, Matthew Woodcraft, May, Mernon, Metamorph, Metric, Mhym, Michael Daly, Mindmatrix, Mpadi, Mms, Modster, Mokendall, Morgenil, Motor, Mountaineer historian, Moerton, Mrwojo, Ms2ger, Mwasheim, Nabokov, Naddy, Neijlt, Netoholic, NevilleDNZ, Night Gyr, Nikai, Ninly, Nitpicker06, Novalis, Oliver Crow, Optikos, Owen, Parallelized, Persan, Peter Flass, Phe, Phil Frost, PleaseStand, Pmsyyz, Pnm, Poor Yoric, Ppl, Prosfilaes, Proteus, QmunkE, RTC, Ranveig, Reyk, Rich Farmbrough, Rjwilmsi, Rocastelo, Rory096, RoseParks, Rosen, Rrreese, Runtime, Rurus, Ruud Koot, SDC, SEI Publications, SLi, Sangwine, Seajay, Sealican, Seanol, Sen Mon, Senator2029, Sfx42, Shadowjams, Shuffdog, Sonicsuns, Soumyasch, SpNeos, Staeker, Sttaft, Suruna, Tassedethe, Taw, Technobadger, Tedickey, Template namespace initialisation script, The Centipede, The Thing That Should Not Be, Thue, Thumperward, Tobias Bergemann, Tompsc, Topfix, Torla42, Uffe, Ultimus, Urhixidur, Utcursch, UtherSTRV, Val42, Vinhtrantran, Wavelength, Wernher, Who, Wickorama, Wiki Tesis, Wikibob, Wikid77, Wikikrs, William Allen Simpson, Wknigh94, Woohooikit, Worrydream, Wshun, X7q, Xyb, Yandman, Yoninah, Ysangkok, 326 anonymous edits

C++ Source: <http://en.wikipedia.org/w/index.php?oldid=519902134> Contributors: -Barry-, 12.21.224.xxx, 1exec1, 223ankher, 28bytes, 4jobs, 4th-otaku, 7DaysForgotten, @modi, A D Monroe III, A Jav, A.A.Graff, AIOS, ALOTOFFTOMATOES, AMackenzie, AOC25, ARSchmitz, ATHing, ATren, Aandu, Abdull, Abi79, Abledsoe78, Abolitionht, Access Denied, Adam12901, Addihockey10, Adi211095, Adorno rocks, Adrianwn, Ae-a, Aesopos, AgadaUrbanit, Agasta, AgentFriday, Ahmadmashhour, Ahoerstemeier, Ahy1, Akeegazooka, Akersmc, Akhristov, Akihabara, Akuyume, Alan D, AlbertCahalan, AlecSchueler, Aleenf1, AleksanderVatov, AlexKarpman, Alex, Alexius08, Alexxon, Alhoori, Alieken, AlistairMcMillan, Allstarecho, Alpha Quadrant (alt), AltiusBimm, Alxeedo, AnAccount2, AnOddName, Andante1980, Andre Engels, Andreaskem, Andrew Delong, Andrew1, AndrewHowse, AndrewKepert, AndyLuciano, Angbor, AngelOfSadness, Angela, Anoko moonlight, AnonMoos, Anonymous Dissident, Antandrus, Aparna.amar.patil, Apexoservice, Apokrytaros, Arabic Pilot, Aragorn2, Arcadie, Arctic.gnome, Ardonik, ArglebargleIV, AshishDandekar7, Asimzb, Astronautics, Atjesse, Atlant, Auntof6, Austin Hair, Autopilot, Avoran, Axecution, AxelBoldt, BMW Z3, Baa, Babija, Babjisit, Backdoor2world, Bahram.zahir, Barek, Barkha dhamechai, Baronnet, Bart133, Bartosz, Bdragon, Belem tower, BenFrantzDale, Benhocking, Bent00, Beowulf king, Bevo, Beyondthislife, Bfzhao, Bhahas, Biblioth3que, Bigk105, Bill gates69, Billj321, Billyoneal, Bineet, Bkl, Blaisorblade, Bluenoose, Bluezy, Bobazoid, Bobblewik, Bober71, Bobo192, Bobthebill, Bodkinator, Boffob, Boing! said Zebedee, Bomarrow1, Bonadea, Bongwarrior, Booklegger, Bosko, Bovineone, Brion VIBBER, Btx40, C Labombard, C++ Template, C.Fred, CALR, Cleland, CPMcE, CRGreathouse, CTZMSC3, CWY2190, Caesura, Caiaffa, Callek, Caltas, Can't sleep, clown will eat me, CanisRufus, Cap'n Refsmmat, Capi crimm, CapitalR, Capricorn42, Captainhampton, Carabinieri, Carbon editor, Card Zero, Carlson-steve, Carnivorousfungi, Catgut, Cathack, CecilWard, Cedars, CesarB, Cetinsert, Cfeet77, Cfim001, Cgranade, Chaos5023, CharlotteWebb, Chealer, ChessKnight, Chocolateboy, ChrisGaultier, Chrisandtaund, Christian List, Chuq, Cburke, Clay, Cleared as filed, Closedmouth, Clsdennis2007, Coasterlover1994,

Code-Analysis, Cometsstyles, Comperr, Conversion script, Coolwanglu, Coosbane, Coq Rouge, CordeliaNaismith, Corrector7007, Corti, Cowsnatcher27, Cpiral, Craig Stuntz, Crashburn2007, Crotmate, Csmaster2005, Ctua2485, Cubbi, Curb Chain, Curps, Cwitty, Cybercobra, Cyclonenim, Cyde, CyrilleDunant, Cyrus, DAGwyn, DJ Clayworth, DVD R W, DVdm, Dallison999, Damian Yerrick, Damien.c.sadler, Dan Brown123, Dan Harkless, Dan100, Danakil, Danger, Daniel Earwicker, Daniel.Cardenas, DanielNuyu, Danim, Darestium, Dario D., DarkHorizon, Darkmonkeyz321, Darolew, Dave Runger, Daverose 33, David A Bozzini, David H Braun (1964), Davidking20, Dawn Bard, Dawnseeker2000, Dech888, Dcoetzee, Decltype, Deflective, Deibid, Delirium, Delldot, Demonkoryu, Denelson83, DerHexer, Derek Ross, Deryck Chan, DevSolar, DevastatorIIC, Devengosalia514, Dewritech, Dibash, DigitalMediaSage, Dirkbb, Discospinster, Dlae, Dlfskj;lskj, Dmharvey, Dnstroy, Dogcow, DominicConnor, DonelleDer, Donhalcon, Doofenschmirtzevilinc, DoubleRing, Doug Bell, Dougjih, Doulos Christos, Drangon, Drewcifer3000, Drmrgrv, Dylnuige, Dysprosia, Dystopianvirus, E Wing, Eskog, Eagleal, Ebisher, Eco84, Ecstaticid, Ed Brey, Editorplus123, Edward Z. Yang, Eelis, Eelis.net, Ehn, Elitewinner, Elliskev, Elysdir, Enarsee, EncMstr, Enerjazzer, EngineerScotty, Entropy, Eric119, ErikHaugen, Esanchez7587, Esben, Esmito, Esrogs, Eternaldescent08, Ethan, EvanED, EvanH2008, Evice, Evil Monkey, Ewok18, Excialr, FW4NK, Fa2sA, Face, Facoreead, Faithlessthewonderboy, Faizni, Falcon300000, Fanf, Fashionslide, FatalError, Favonian, Fistboy, Fizzackerly, Flamingantichimp, Flewis, Flex, Fluffernutter, Flyingprogrammer, FrancoGG, Franl, Frocknfurnture, Frecklefoot, Free Software Knight, Freshenees, Fritzpoll, Ftbhrygvn, Furby100, Furrykef, Fuzzybyfe, Fyyer, GDallasmore, GENtLe, GLari, Gareth Griffith-Jones, Gaul, Gauss, Geoharee, Gene.thomas, Gengiskanh, Georg Peter, Giftlite, Gil mo, Gildos, Gilgashem, Gilliam, Gimili2, Glacialfox, God Of All, Gogo Dodo, GoingBatty, GoodSirJava, Graue, Greatdebtor, GregorB, Gremagor, Grenavitar, Grey GosHawk, Grigor The Ox, GroovySandwich, Gsonnenf, Gusmoe, Gwern, Gwames, Hairy Dude, Hakkinen, Hamtechperson, Handheldpenguin, Hans Bauer, HappyCamper, Harald Hansen, Harinisanthosh, Harryboyles, HebrewHammerTime, HeikoEvermann, Hemanshu, HenryLi, Herover, Hervegirod, Hetar, Hex, Hgfernan, HideandLeek, Hihaihia474, Hiraku.n, Hmainas, Hobartimus, Hogan500, Horselover Frost, Hoss7994, Hu, Hu12, HueSatLum, Husond, Hyad, Hyperfusion, I already forgo, ISO LoveHer, Iamini9191, Ibmular, Ibroadfo, Iliealldaylong, Imc, Immunize, InShanee, Innocent, Insanity Incarnate, Intangir, Intelati, InvertedSaint, Ipnoneorange, Ipsign, Iridescence, Iridescent, Irish Souffle, Ironholds, Isaacl, Ifxfd64, J Casanova, J Di, J-A-V-A, J.Dong820, Jdelanoy, JForget, JNighthawk, Jackelfive, Jafet, Jarble, Jaredwf, Jason Quinn, Jatos, JavieraCastillo73, Javierito92, Jawed, Jayaram ganapathy, Jdent29, Jdowland, Jeff G., JeffTL, Jeltz, Jerry teps, Jerryobject, Jeshan, Jesse Viviano, JesseW, Jfmantis, Jgamer509, Jgrahn, Jgroen, Jh51681, Jim1138, Jimsvs, Jizzbug, Jk2q3jrkls, Jin, Jnestorius, Johnclci, Johnsolo, Johnunji, Jojo666, Jok2000, Jon701, Jonathan Grynspan, Jonathanischoice, Jone1, Jonmon6691, Joren, Josh Cherry, Juliano, Julianlecomte, Junkyboy55, Jyotirmay dewangan, K3rb, KJK:Hyperion, KTC, Kaimaison1, Kajausudhakarababu, Kalanaki, Kallikanzarid, Kanakid200, Kangaroopower, Kapil87852007, Kashami, Kate, Keilana, Kentij, Khyn Chaur, Kifcaliph, King of Hearts, Kiuu, Kkm010, Klassobanerias, Kleusus, Klilidiplomus, Kmern, KnowledgeOfSelf, Kogz, Kooky, Korath, Korval, Koyaanis Qatsi, Krelborne, Kri, Krich, Krischik, Kristjan.Jonasson, KryptonX, Ksam, Kuru, Kusunose, Kwamikagami, Kwertii, Kxx, Kyle2^32-1, Kyleahampton, Landon1980, Larry V, Lars Washington, Lastplacer, Le Funtinae Frankie, Lee Daniel Crocker, LeinadSpoon, Lepreshaun, Leszek Jaćzuk, Liao, Liftern, Lightmouse, Ligulem, Lilac Soul, Lilpny6225, Lir, Liuijiang, Lkdude, Lloyd Wood, Loaderman, Locos epaix, Logixoul, Lost.goblin, Lotje, Lovetinkle, Lowellian, Luks, Lvella, Lysander89, MER-C, Mabdul, Machecku, MadCow257, Mahanga, Maheshchowdary, Male 1979, Malful, Malhonen, Malleus Faturuman, Man11, Manjo mandrava, Manofabuledog, Manolis69, MarSch, Marc Mongenet, Marc-André Abbrock, Marcelo Pinto, Mark Foskey, Marktillinghast, Marqmiike2, Martarius, Masterkillia, Materialscientist, Mathrick, Mato, Matt.forestpath, MattGiuca, Matthewp1998, Matthieu fr, Mav, Mavarok, Max Schwarz, Maxim, Mayank15 5, Mbecker, Mccoy, Mcorazio, Mcstrother, Mean as custard, Meccanism, Mellum, MeltsBanana, Mentifisto, Mephistophelian, Mequa, Meshach, Metamatic, Meters, Methucb, MetsFan76, Mfwitten, Mhadoks12, Mhnin0, MichaelJHuman, MichaelTarpaley, Micphi, Mifter, MihaS, Mikademus, Mike Rosoft, Mike Van Emmerik, Mike92591, Mikrosan Akademija 7, MilesMi, Millahma, Mindmatrix, Minesweeper, Minghong, Minimac, Mipadi, Miquademus, Miracleworker5263, Miranda, Mirror Vax, Mistersooreams, Misuo, Mqujinn id, Mkarlesky, Mkcmkc, Mkhani3189, Mmeijeri, Mmugundham, MoAgnome, Moanzhu, Modif, Mohamed Magdy, Mole2386, MoreNet, Morgankevinj, Morwen, Moxfyre, Mptb3, Mr MaRo, Mr.GATES987, MrJeff, MrOllie, MrSomeone, MrX, MrJeff, MrWres95, Ms2ger, Muchness, Mukis, Muneermpballa, Muralive, MustafaenS, Mxni, My76Strat, Myasuda, Mycupl, Mystic, N111111KKKKKKKoooodo, Naddy, Nanshu, Napi, Nasa-verve, Natdaniels, Nathan2055, NawlinWiki, Neilc, Neurolysis, NevilleDNZ, Newsmen, Nfcopier, Nick, Nicsterr, Niels Dekker, Ninly, Nintendude, Nirdh, Nisheet88, Nixdorf, Nixeagle, Njaard, Nima wiki, Nohat, Noldoaran, Non-dropframe, Noobs2007, Noosentaal, Northernhenge, Ntisimp, ORBIT, OUDave4m, Odddee323, Oddity-, Odinjobs, Ohnoitsjamie, Ojuice, OldakQuill, Oleg Alexandrov, Oliver202, Oneiros, Orderud, Ouraqt, OutRIAAge, OverlordQ, OwenBlacker, Ozzmosis, Paddu, Padmaja cool, Pak21, Panarchy, Pankajwillis, ParallelWolverine, Paul Stansifer, Paul evans, Paulcmnt, Paulius2003, Pavel Vozenilek, Pawanindia2009, Pbroks13, Pcb21, Pde, PeaceNT, Pedant17, Peruvianlrama, Peteri, Peteturte, Petiati, Pgk, Pharaoh of the Wizards, Pharus, Phil Boswell, Philip Trueman, PhilippWeissenbacher, Phoe6, Pi is 3.14159, Pinethicket, Pit, Pizza Puzzle, Plasticup, Popigopi, Poldi, Polluxian, Polonium, Polyamorph, Poor Yorick, Potatoswatter, Prashan08, PrincessofIlyr, Prohlep, Protokon, ProvingReliability, Pt, Pytxs, Punctilius, QECUB, Quadell, QuentinUK, Quinsareth, Quuxplusone, Qwertys, R3m0t, R4rtutorials, REggert, Raghavkp, RainbowOfLight, Rallias, Ravisanarkv, Rbonvall, Rdsmith4, Reach Out to the Truth, RedWolf, Redhanker, Reffyy, Rehabe, Reinderien, Reisio, Remember the dot, Requestion, Rethnor, RexNL, Rgb1110, Rich Farmbrough, Richard Simons, Richard24simon, Rijurekh, Ritualizer, Rjrock, Rjwilmsi, Roadrunner, Robdumas, Robertd, Robo Cop, Rockfang, RodneyMyers, RogueMomen, Rold50, Romark, Ronhjones, Ronnyim12345, Ronyclaus, Root@localhost, Rosive, Rossami, RoySmith, Royote, Rprpriya, Rror, Rrif, Rursus, Ruud Koot, Ruy Pugliesi, Ryan Norton, RyanCross, Ryty01, SJP, STL, Sachin Joseph, Sadday, Sae1962, Saigopalrocks, Saimie, Salesvery1, Samthedude55, Samuel, Sasha Slutsker, Sbsolo, Sbvb, SchfrityThree, Schirali, SchnitzelMannGreek, Schumi555, ScoPi, Scoops, Scorp.pankaj, Scottlemke, Scythe33, SebastianHelm, Sebastiangarth, Sebor, Sentense12, Seraphim, Sergei, Sev2013, Sfxdude, Sg227, Shadowblade0, Shadowjams, Shaurya44, Shawnc, SheffieldSteel, ShellCoder, Shinjiman, Shiv narayan, Shrike, Shriram, Sidhanta, Sigma 7, Silivrenon, Simetrical, Simon Brady, Simon G Best, SimonP, Simplas, Sinternational, Sirex98, Sirupuramrk, Sishgupta, Sitetchief, Skew-t, Skizzik, SkyWalker, Sl, Sleep pilot, Sligocki, Slothy13, Smsarmad, Smyth, Snaex920, Snettel, Snigbrook, Snowlow, Sohmc, Sokari, SomeRandomPerson23, Sommers, Sowsneak, Spaz man, Spiel, Spitfire, SplinterOfChaos, Spokestrip EMP, SpuriousQ, Sra.nasir, Stanthejeep, SteinBDj, Stephan Schulz, Stephen, Steve carlson, Stevenj, StewartMH, Stheller, Stoni, StoptheDatabaseState, Strangnet, Strcat, Strangle, Stuarcift, Style, Suffusion of Yellow, Supertouch, Suppa chuppa, Surv1v411st, Sutambe, SvGeloven, Svick, Swalot, SwisterTwister, Sydius, T, T0pem0, T4bits, TCorP, THEN WHO WAS PHONE?, Takis, TakuyaMurata, Tattema, Tbleher, TeaDrinker, Technion, Tedickey, Template namespace initialisation script, Tero, Tetra HUN, TexMurphy, Tgeairn, Tghramann1, The 88th Avatar, The Anome, The Inedible Bulk, The Minister of War, The Nameless, The Thing That Should Not Be, TheDeathCard, TheIncredibleEdibleOompaLooppa, TheMandarin, TheNightFly, TheSuave, TheTim, Theatrus, Thebrid, Theda, ThematrixV, Thiagomael, Thumperward, Tietew, Tifeo, Tigerbombs8, Tim Starling, Tim32, TingChong Ma, Tinus, TitaniumCarbide, Tobias Bergemann, Toffile, TomBrown16, TomCat2800, Tomalak geretkal, Tommy2010, Tompsc, Tomshutterbug, Tony Sidaway, Topsfield99, Torc2, Tordek ar, Toussaint, Tpbadbury, Traroth, Trevor MacInnis, TreyHarris, Treols Arvin, Trogdor31, Trusilver, Ts4z, Tsclocum, Turdboy3900, Turian, TuukkaH, Ubardak, Umapathy, Unendra, Ungahbunga, UnicornTapestry, Urhixidor, Urod, UrsaaFoot, Useight, Userabc, UtherSRG, Utapnistin, VTPG, Val42, Van fisted, Vanished user 99034jfoiasjq2oirhsf3, Vchimpanzee, Vincenzoromanico, Vinci0008, Viperez15, VladV, Vladimir Bosnjak, Vrenator, Wangi, Wavelength, Wazzup80, Werdn, Westway50, Whalelover Frost, Who, Wickorama, WikHead, Wikidemon, Wikidrone, Wikipendani, Wikiwonky, Willbennett2007, Wilson44691, Winchelsea, Wj32, Wknight94, Wlievens, Woohookitty, WookieInHeat, Writ Keeper, Wsikard, Witb435, Witmittch, XjamRastafire, Xelnos21, Xeresnine, Xoadnotent, Yamla, Yankees26, Yashyk, Yboord028, Ybungalbil, Ycubed100, Yoshirules367, Ysangkok, Yt95, Yurik, ZacharyM001, Zek, Zed toocool, Zenohockey, Zigmars, Zlog3, Zoe, Zr2d2, Zrs 12, Zundark, ZungBang, Zvn, Ævar Arnfjörð Bjarmason, ABΓΔΕΖΗΟΙΚΑΜΝΞΟΠΣΤΥΦΧΨΩ, Σ, Александър, Пеънца, А্যামън, 无名氏, 2281 anonymous edits

Duck typing Source: <http://en.wikipedia.org/w/index.php?oldid=518606379> Contributors: Ahkilinc, Alansohn, Amourfou, Andy Dingley, Austin Hair, Barakafrit, Blbfish, BenFrantzDale, Bkell, Bluenoose, Bohumir Zamecnik, BostonMA, Breakpoint, Brianski, Bvbellomo, CWenger, CesarB, Chammy Koala, Charivari, Chip Zero, Chipuni, ChuckEsterbrook, Cjoev, Colonies Chris, Conrad.Irin, Cybercobra, Damian Yerrick, Danchr, Daurnimator, Dbabbitt, Devnevyn, Dk pdx, Doradus, Dougher, Dougweller, Dqd, Dreftymac, Dwememouth, Egglizard, Ethelhael, Evildeathmath, FatalError, Finell, Finkded, Fred Bradstadt, Fredrik, Freshenees, Furrykef, Geekner, Geonick, Georgesawyer, GoingBatty, Greatestrowever, Hanenkamp, Hdante, HereToHelp, Hervegirod, Hu 8.5, Håvard Fjær, Iapetus, Iggywmangi, Ikc-bana, Indl, InverseHypercube, JMTwlk, Jake Wartenberg, Javierito92, Jcarneian, Jerryobject, JimD, Jmmbatista, JonathanBaldwin9182, Joren, Joswig, Jurgen Hissen, Jystein, Krallja, Ksn, Kwizy, Liangent, Licon, Mal4mac, MartinPoulter, Marudubshinki, MatthieuWiki, Meelar, Microsoftfb, MikeTA, Nabla, Nagy, Nealcardwell, Nicolas de Marqué, Njd27, Obankson, Ohnoitsjamie, Oluijes, Ore4444, Paddy3118, Pavel Vozenilek, Phunehhe, Piet Delpot, Pol430, Prybaltowski, Pyanton, Rettetast, ReyBrujo, Rich Farmbrough, Ricky Clarkson, Ricvelozo, Rjwilmsi, Robennals, RoySmith, Rschmertz, Sagaciousuk, SebastianGarth, Sedimin, Shellreef, Sietse Snel, Sikon, SimonTrew, Sliwers, Snaex920, Solidpege, Spaceyguv, Spidermario, StanfordProgrammer, StephenFalken, Sunetos, TakuyaMurata, Tangotango, Tarclieri, Technobadger, Terron, The Anome, The Stickler, TobiasHerp, Toddfast, Tom Morris, Toobab, Torzsmokos, Tukuhka, Valodzka, Van der Hoorn, Visor, Whitepaw, WriterHound, Yahalom, Yakushima, Yath, ZacBowling, Zennehoy, Zenthing, ZeroOne, "zer0dyne\$_Elfrie", Александър Чуранов, 253, שְׁנִי anonymous edits

Ruby (programming language) Source: <http://en.wikipedia.org/w/index.php?oldid=520258826> Contributors: -Barry-, .:Ajvol:, .:Ajvol:, 217.98.151.xxx, 217.99.96.xxx, A3RO, A5b, Abhishekandan25, Aceofspades1217, Agusti, Ajpeters, Al Hart, Alexkon, AlisonW, AlistairMcMillan, Amecaf1, Anastrophe, Andre Engels, Andrevan, Andyabides, AnibalRojas, Anna Frodesiak, Anon lynx, Anthony, Antonio Cangiano, Apotheon, Arash Hemmat, Archanamiya, Arcnova, ArlenCuss, Aruton, Ashleyjoyce, Atanamir, Atlant, AurakDraconian, Austin Hair, AxelBoldt, Azatotha, Baator, Bailey.d.r, Barte, Beinsane, Beland, Ben White, Bender235, BenjaminOakes, Betacommand, Bevo, BigFloppyDonkey1234, Bkkbrad, Black Falcon, Blade Hirato, Bramazuyer, Bobdc, Bovlb, Bowman, Bravestarr, Brion VIBBER, Bspeakmon, BurnDownBabylon, Busbeytheelder, CRGReathouse, Cadr, CatPaterson, Carlj7, Cbenz, Ceejayoz, Centrx, Chasingsol, Chealer, Chipuni, ChrisGualtieri, Chubanu Baba, CiudadanoGlobal, Closedmouth, Cmol, Colonies Chris, Connally, Conversion script, Coop, Corti, Countchoc, Craig Stuntz, Crazycomputers, Crenner, Cwolfsheep, Cybercobra, D'oh!, DPoon, Daekharel, Daev, DanKohn, Danakil, DanielAmelang, Danshep79, DarkseidX, Dasch, David from Downunder, Dcoetzee, DeadlyAssassin, Deep.deep, Demonkoryu, Derek Ross, Derex, Destynova, Dfrankow, Diego.viola, Digitalsurgeon, Dinomite, Discospinster, Docreddi, Dominikhonnef, Donhalcon, Drbrain, Dreftymac, Drj, Drmies, Droob, Drz, Ds13, Dublinclontarf, Duplicity, Dv, Djvjones, Dyspepsia, ELEMIV, Edc, Eipupz, Evolve2k, Evolvingjer, Excirial, FF-Wonko, FF2010, Fabio.kon, Faisal.akeel, Faya, Ffangs, Flankk, Flankk2, Flying sheep, Fniesen, Fraggle, Francis Ocoma, Frau Holle, Fredrik, Friendyoc, Fukumoto, Furrykef, Gaius Cornelius, Galactic Hitchhiker, Garkbit, Ge9580, GeneralAntilles, Geoffr, GeorgeMoney, Georgesawyer, Giftlite, Gmaxwell, Gobonobo, Gogo Dodo, Gpvs, GrEp, Graue, GregorB, Gregwebs, Gronky, Guaka, Gudeldar, Gunark Gustavb, Gwern, HDCase, Hairy Dude, Hajhouse, Halo, Hannes Hirzel, Hao2lian, Hashar, Headius, Heja2009, HenkeB, HenryLi, Herraotic, Hervegirod, Hessammehr, Heyseuss, Hilltopperpete, Hirzel, Homeobocks, Hu2hugo, Huw Collingbourne, Hyad, Hydrox, I already forgot, Ianblair23, Ianozvald, Igouy, Ioannes Pragensis, Ispign, Isfish, Islanes, J.delanoy, JGXenite, JackH, Jackliddle, James Britt, Jamieorc, Jason.grossman, Jatkins, JavaTenor, Jazzwick, Jcubic, Jedediah Smith, Jeltz, Jeremy Devenport, Jerryobject, Jessecooke, Jessemerriman, Jikanbae, JimJavascript, Jm34harvey, Jmorgan, Joeljkp, Jeroite, Jgoloran, Jorge Stolfi, Joshua Scott, Jowa fan, Jrtayloriv, Klee, KKL, KKong, KMeyer, Kain2396, Kaster, Kbh3rd, Kengo.nakajima, KevinScott, Khatharr, King of Hearts, Kl4m, Kl4m-AWB, Kmactane, Knutux, Kokot.kokotisko, Koolabsol, Kopf1988, Korean alpha for knowledge, Ksn, Kusunose, Kvdeveer, L Kensington, Lasah, Laughing Man, LazyEditor, Lee Daniel Crocker, LethargicParasite, Li-sung, LiDaobing, Lipedia, LittleDan, Logi, LordRdm, Lulu of the Lotus-Eaters, Lupinoid, MER-C, MGPCoe, MIT Trekkie, Macy, MarSch, MartinRinehart, Marudubshinki, Masiarek, Maslin, Mattsenate, Mattemp, Matusz, Maximimax, McSly, Mcaruso, Mcorazao, Mcses, Mernen, Merrells, Metamatic, MHagerman, MicahElliott, Michaelneale, Michaelpb, Michalis Famelis, Micropolygon, Midnight Madness, Mignon, Mindmatrix, Minesweeper, Mipadi, Mithaca, Mkarlesky, Mnemo, Mohawkjohn, MojoX, MonkeeSage, Morning277, Mrbill, Mrego, Mujaki, Mxn, MySchizoBuddy, Nanshu, Neilc, Neuronutron, Nick Nolan, NickelShoe, Nigelj, Nikai, Nmgagedman, Nonpr3, Northgrove, Notheruser,

Ohnoitsjamie, Oli Filth, Oneiros, P69, PGSONIC, Panglossa, Paolino, Paraphelion, Paxinum, Pdq, Perle, PeteVerdon, Peterl, Pgant002, Phil Boswell, Pilaf, Pmcn, Pne, Polveroj, Poor Yorick, Pradameinhoff, Proofreader77, PsyberS, PuerExMachina, Qwertyus, R3m0t, Radnam, RandalSchwartz, Ratioincate, Raypereda, Rbonvall, RedWolf, Renku, RenniePet, Rich Farmbrough, Rickjelleg, Ricvelozo, Rjwilmsi, Rob*, Robert Dober, Robertd, Robertwharvey, Robhu, RockMFN, Rogerdpack, RolandH, Ronark, Rule.rule, Ruud Koot, RyanJones, Saikumarganj, Salvan, Sander Säde, Scientus, Scott Paeth, SeanPage31, Seaphoto, SevanMilis, Shadowfirebird, Shadowjams, Sharcho, Shipmaster, ShotaFukumori, SimpleBeep, Sindhu sundar, Sjc, SkyWalker, Sligocki, Slocombe, SlurpTheo, SnappingTurtle, Snarius, Sofy.Basir, Sonjaaa, Soumyasch, Sp0ng, SpeedyGonsales, Spivo, Spitfire, Srittau, Ssd, St.daniel, Stevemidgley, Stevietheman, Stickmangumby, Stillwaterising, Storm Rider, Strangnet, StudioFortress, Stygian23, Su30, Sunnymoriah, Supermatique, TJRC, Tarquin, Taw, Tb, Tcooke, Template namespace initialisation script, Tfgbd, The Earwig, The Extremist, The Thing That Should Not Be, The wub, TheAstonishingBadger, TheCriticizer, TheMathinator, TheOtherJesse, ThePCKid, Thomas Linder Puls, Thumperward, Timosa, Tlesher, Tmcw, Toby Bartels, Tom Morris, Tomchen1989, Tony Sidaway, Trappist, Tthorley, Turnstep, Ulric1313, Unforgettableid, Unixphil, Usien6, Van der Hoorn, Vanyo, WRK, Wavelength, Wegesrand, Wemagor, Wertuose, WhatisFeelings?, Whayworth, Wickorama, WikiPuppies, Wikievil666, WikiVan13, Will Beback Auto, Windharp, Winterheat, Wisdomgroup, Wiz, Wrp103, Wwwolf, Xavexgoem, Yahoolian, Yamamoto Ichiro, Yath, Youssefyan, Yserbnaaj, Yurik, Ziplux, Zootm, Zudduz, Σ, Александър, 899 anonymous edits

Concurrency (computer science) *Source:* <http://en.wikipedia.org/w/index.php?oldid=518438763> *Contributors:* 2ndMouse, AS, Abdull, Aldinuc, Allan McInnes, Altenmann, Archelon, Autopilot, BehnazCh, Brick Thrower, CarlHewitt, Comestyles, Comps, Cstb, Danakil, Danielheres, Eskimosweeproll, EvanProdromou, Ghakko, Good Olfactory, GregWoodhouse, Heyjoy, HyperFlexed, Jab843, JonHarder, JonathanMayUK, KevinCarlson, Kku, LouScheffer, M4gnun0n, Mathnerd314159, Matthewrbowker, Mdd, Michael Hardy, Mion, MrX, Neutral current, Nuno Tavares, Ohiostandard, ParallelWolverine, Partialorder, Piet Delpot, Pinclar, Rawafmail, Reedy, Rjwilmsi, Ruud Koot, Samsara, Signalhead, SimonD, Stephan Leeds, Tarcieri, TheProgrammer, Timwi, Vald, VictorAnyakin, Vonkje, Wiretse, Лев Дубовой, 68 anonymous edits

Erlang (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=519941655> *Contributors:* ::Avjol, 1234456556BK, 1exec1, 2001:6F8:1C8D:0:F24D:A2FF:FEF8:D03C, 209.75.42.xxx, 4th-otaku, Adamncasey, Adrianwn, Akavel, Allan McInnes, Amire80, AmritTuladhar, Andreas.hillqvist, Andyjsmith, Anna Frodesiak, AnomMoos, AntiRush, Antonielly, Argv0, Arn7, Ashley Y, AssociatOr, AutumnSnow, Barkjon, BenBaker, Bender235, Bosmon, Callidior, Camw, Carnildo, ChasRMartin, ChrisGuatieri, Conversion script, Creativetechnologist, Cromain, Csl77, Cybercobra, Danakil, Darklich14, David Gerard, David Wahler, DavidDouthitt, Decltype, Defza, Demi, DennyAbraham, Dognova, Dgpop, Dlambert, Dmitriid, Doradus, Drbrenzjev, Drewnoakes, Ds13, Dysprosia, Ebraminio, Ejabberd, Eliasen, Emurphy42, Erik Cornelisse, Erlang 1, Erlang.consulting, Etz Haim, Faisal.akeel, Fanf, FatalError, Fikusfail, Foobaz, Frap, Frecklefoot, Fudoreaper, Furykef, Gaius Cornelius, Gf uip, Giardante, Gleber.p, Gparalikidis, Gpierre, Graham87, Greenrd, GregorB, Grimboy, Grshiplett, Gwern, H2g2bob, Halfacanuck, Hcs42, Hdante, Hsk0114, Ignatz, Intgr, Iridescent, JLD, JLaTondre, Jakub Mikians, James Hague, JamesBWatson, Janko.stefancic, Jasontrost, Jeltz, Jerryobject, JohnnyBjorn, Jonasagundes, JonathanFeinberg, Jonnabuz, Justin W Smith, JustinSheehy, Karvendhan, Kellen, Keving, Kibwen, Korpo, Krallja, Kricxjo, Kuszi, Kvduveer, Ledomedlem, Levin, Lihaitao, Liujiang, Loopkid, Lproven, Luke.randall, M7, Maciekwar, Mapfn, Mariehuynh, Markpeak, Martinl, Marudubshini, Masharabinovich, Matt Crypto, Matthewdunsdon, Matthewwokeefe, Maxim.kharchenko, Maxlapshin, Maxsharples, Memming, Miketomasello, Misterscupcake, Moltonel3xCombo, Msbmbs, Mwarren.us, NaturalBornKiller, Neelix, Neustradamus, Niten, Nixdorf, Noliver, Nyco, Ohnoitsjamie, Palfrey, Papppfaffe, Pauli133, Pavel Voznenlik, Peepedia, Peet74, Peter S., Pgant002, Piet Delpot, Potto007, Prof3ta, Qiangguo, QuaA, Quadell, Ravidgemole, Renku, Richardhenwood, RickBeton, Riffic, Rklophaus, Rogper, Rohan Jayasekera, Ronewolf, Ronz, Royote, Ruud Koot, Rwalker, Ryzel, Salamangero, Sandos, Sanspeur, Sapien2, Sbierwagen, ScierGuy, Sclm, Sedrik666, ShelfSkewed, Stefan Udrea, Stevenj, Stewartadcock, Stpter, Svick, Tarka, Taw, Teddk, The Anome, Tim Iverson, Tobias Bergemann, Tobias382, TobinFricke, ToddDeLuca, Toniesner, Traroth, Twn, Ultramandk, Uninverted, Untalker, Vectro, Vikreykja, Vocaro, Vroo, Wdkrnl, Xan2, Yakushima, Yaronf, Yoursql, Zetawoof, Zvar, 385 anonymous edits

D (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=520062460> *Contributors:* -Barry-, AThing, Abledsoe, Acf, Akel Desyn, Al E., AlanUS, Andareed, Andrew Delong, AnonMoos, Anonywiki, Ascánder, Atanamir, Attilios, Baxissimo, BazookaJoe, Bender235, Bhny, Black Falcon, Bluemoose, Borgx, Born2x, Boshomi, Bovineone, BreachReach, Bunnyhop11, CRGreathouse, CanisRufus, Catskul, Ceyockey, Chinju, Cic, Clickingban, Cogiat, Cryptic, CyberShadow, CyberSkull, Cybercobra, Damicatz, DanielKeep, Davidhorman, Deewiant, DelphinidaeZeta, Derbeth, Derek Parnell, Dfasdfasdfs121212, DiThi, Diberri, Dissident, Dualathlon, Edam, Eliashc, ErikHaugen, Eyu100, FatalError, Foucher, Frau Holle, Frecklefoot, Fubar Obfusco, Furykef, Fwend, GNRY09, Gaiia Octavia Agrippa, Garyzx, Ghettoblaster, Gilgamesh, Glacialfox, Greenrd, Greg Tyler, Hairy Dude, Hasan aljudy, Hemanshu, Hfdkh, Holy-foek, IamOz, IcyT, Igouy, Int19h, Intgr, Itai, J JMesserly, JWWalker, Jarble, JaschaWetzel, Jeff02, Jerryobject, Jratt, Kate, Kenyon, Khazar, Kinghaji, Klestes, Komitsuki, L Kensington, Leaflard, Lmat, MD87, MaetSan, Marianoceccowski, Marioxxc, MattGiuea, McLar eng, Melnakeeb, Mex1995, Mgreenbe, MichiGrea, MinorEdits, Mkb218, MrSteve, Mynamraviteja, NapoliRoma, Nascent, Natamas, Nezempin, Neile, Nick, Night Gyr, Northgrove, Ojigiri, Optim, Orderud, Palica, Papav, Patrick-br, Paul August, PaulBonser, Paxwill, Pdq, Phoenix-forgotten, Poccil, Possum, Qef, Quaqqo, Qutezwe, Qwertys, Rainault, Rinick, Rogper, Royote, Rubik-wuerfel, Rursus, Ruud Koot, SMP, ST47, Salvolsaja, Sarav62, Sbisolo, Sealican, SeeSchloss, Serprex, Sigma 7, Sigurdhsson, Simxp, Smjg, SpK, SteinbDJ, StephenJGuy, Stevietheeman, Strait, Suisui, Swalot, Sydneyfong, TallNapoleon, The Inedible Bulk, The Thing That Should Not Be, ThereIsNoSteve, Thumperward, Tifego, TimBentley, Timsheridan, Tobias Bergemann, Tom W.M., Tompsc, Torc2, Tsuji, Tuur, Tweenk, Uman, Umawera, Unknown W, Brackets, Val42, VladimirKorablin, Waldir, Wbm1058, Wickorama, Wootery, Wsriley, Xan2, Xenon325, Xompanthy, Yahoolian, Yuu eo, ZeiP, Zkp0s, 421 anonymous edits

Go (programming language) *Source:* <http://en.wikipedia.org/w/index.php?oldid=519738816> *Contributors:* .p2502, 1exec1, 2620:0:1000:2301:26BE:5FF:FE1D:8906, 2A01:348:6:600:0:0:0:2, Abednigo, Abledsoc78, Aclassifier, Adamstac, Aeiththiet, Ahmed Fatoum, Aidoor, Alexandre Bouthors, Alfredo ougaowen, Am088, ArglebargleIV, Atomician, Banaticus, BarryNorton, Biasoli, Bkkbrad, Bla5er89, Blaisorblade, Blue Em, Boshom, Bounce1337, Brianski, Btx40, Chickencha, Choas, ClaudeX, Curly Turkey, Cybercobra, DAGwyn, DMacks, Daniel.Cardenas, Darxus, Davcamer, Dchestnykh, Divas, Docreddi, Drewlesueur, Egmetcalfe, Elimisteve, Ender409, ErikvanB, Espadrine, Filemon, Fraggle, Fried-peach, Fshahriar, Gerardohc, Glenn, Goplexian, Gracefool, Google, Guy Harris, Gzhao, Happyrabbit, Hervegirod, Hexene, Hmains, Hydrex, Ian13, Isaac B Wagner, JC Chu, JaGa, JamesBrownJr, Jarble, Jerryobject, Jeysaba, JnRouvignac, JonathanMayUK, Jonovision, JorgePeixoto, Juancnuno, Julesd, Kendall-K1, Kinema, Letdorff, Loadmaster, Lopifalko, Lost.goblin, Lt. John Harper, MarsRover, Martijn Hoeksstra, Masharabinovich, Matt Crypto, Mdemare, Melnakeeb, Meltonkt, Michael miceli, Mild Bill Hiccup, Mindmatrix, Mkcmkc, Mu Mind, MuffledThud, Nasnema, Nealmeb, Northgrove, OsamaK, PBS, PatrickFisher, Perey, Petter Strandmark, Pgant002, Pgr94, Philwiki, Prodigus, ProfCoder, Prolog, RKT, Rahulrulez, Raysonho, Robennals, RoodyAlien, Rthangam77, SF007, Set72, Shii, Sigmundur, Slartidan, Soni master, Stefan.karpinski, Stokito, Strcat, Stybn, Svaksha, Sverdrup, Swehack, Syp, TakuyaMurata, Thumperward, Tsuba, Tuggler, Tuxcantfly, Twilsonb, Waldir, Wavelength, Wickorama, WikiLaurent, Wikipelli, Woz2, Wrackey, XP1, Xan2, Yves Junqueira, ZacBowling, Zephyrtronium, உத்தாமுருங்கி, 172 anonymous edits

Image Sources, Licenses and Contributors

Image:Python add5 parse.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Python_add5_parse.png *License:* Public Domain *Contributors:* User:Lulu of the Lotus-Eaters

Image:Python add5 syntax.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Python_add5_syntax.svg *License:* Copyrighted free use *Contributors:* Computer tower, Lulu of the Lotus-Eaters, Nerzhal, Nixón, Red Rooster, Savh, Xander89, 6 anonymous edits

File:Bangalore India Tech books for sale IMG 5261.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Bangalore_India_Tech_books_for_sale_IMG_5261.jpg *License:* Creative Commons Attribution-Sharealike 3.0 *Contributors:* User:Victorgrigas

File:Structured program patterns.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Structured_program_patterns.png *License:* Creative Commons Zero *Contributors:* User:Orion 8

File:Fortran acs cover.jpeg *Source:* http://en.wikipedia.org/w/index.php?title=File:Fortran_acs_cover.jpeg *License:* Public Domain *Contributors:* Boshomi, Closeapple

File:IBM704.gif *Source:* <http://en.wikipedia.org/w/index.php?title=File:IBM704.gif> *License:* Attribution *Contributors:* Lawrence Livermore National Laboratory

File:FortranCardPROJ039.agr.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:FortranCardPROJ039.agr.jpg> *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Arnold Reinhold

File:FortranCodingForm.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:FortranCodingForm.png> *License:* Public Domain *Contributors:* Original uploader was Agateller at en.wikipedia

File:LISP machine.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:LISP_machine.jpg *License:* Creative Commons Attribution-ShareAlike 3.0 Unported *Contributors:* Bdamokos, Garcon, Hydrargyrum, Jszigetvari, Mu301, Nameless23, 3 anonymous edits

File:Loudspeaker.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Loudspeaker.svg> *License:* Public Domain *Contributors:* Bayo, Gmaxwell, Gnosygnu, Husky, Iamunknowm, Mirithing, Myself488, Netnac DIU, Omegatron, Rocket000, Shamugamp7, The Evil IP address, Wouterhagens, 23 anonymous edits

File:Steve Russell-PDP-1-20070512.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Steve_Russell-PDP-1-20070512.jpg *License:* Creative Commons Attribution 2.0 *Contributors:* Joi Ito from Inbamura, Japan

File:John McCarthy Stanford.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:John_McCarthy_Stanford.jpg *License:* Creative Commons Attribution-Sharealike 2.0 *Contributors:* "null0"

File:Cons-cells.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Cons-cells.svg> *License:* Creative Commons Attribution 3.0 *Contributors:* Original uploader was Yonkeltron at en.wikipedia

File:Green check.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Green_check.svg *License:* Public Domain *Contributors:* gmaxwell

File:Red x.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Red_x.svg *License:* Public Domain *Contributors:* Anomie

Image:Lambda lc.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Lambda_lc.svg *License:* Public Domain *Contributors:* Luks, Vlsergey, 1 anonymous edits

File:Haskell-Logo.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Haskell-Logo.svg> *License:* Public Domain *Contributors:* Thought up by Darrin Thompson and produced by Jeff Wheeler

File:oop-uml-class-example.png *Source:* <http://en.wikipedia.org/w/index.php?title=File:Oop-uml-class-example.png> *License:* GNU Free Documentation License *Contributors:* Original uploader was Esap at en.wikipedia

File:Wikibooks-logo-en.svg *Source:* <http://en.wikipedia.org/w/index.php?title=File:Wikibooks-logo-en.svg> *License:* logo *Contributors:* User:Bastique, User:Ramac et al.

File:Javascript icon.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Javascript_icon.svg *License:* GNU Lesser General Public License *Contributors:* Lupo

Image:Ada Lovelace 1838.jpg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ada_Lovelace_1838.jpg *License:* Public Domain *Contributors:* William Henry Mote

File:BjarneStroustrup.jpg *Source:* <http://en.wikipedia.org/w/index.php?title=File:BjarneStroustrup.jpg> *License:* GNU Free Documentation License *Contributors:* -

File:Ruby logo.svg *Source:* http://en.wikipedia.org/w/index.php?title=File:Ruby_logo.svg *License:* Creative Commons Attribution-Sharealike 2.5 *Contributors:* Yukihiro Matsumoto, Ruby Visual Identity Team

Image:Dining philosophers.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Dining_philosophers.png *License:* unknown *Contributors:* User:Avatar, User:Bdesham

File:Erlang logo.png *Source:* http://en.wikipedia.org/w/index.php?title=File:Erlang_logo.png *License:* Trademarked *Contributors:* Butko, Kyro, WikipediaMaster

License

Creative Commons Attribution-Share Alike 3.0 Unported
[//creativecommons.org/licenses/by-sa/3.0/](http://creativecommons.org/licenses/by-sa/3.0/)