Handbook of Computer Programming and Hardware Description Languages

Justine Forney

Leta Ivory

First Edition, 2012

ISBN 978-81-323-1275-8

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

Table of Contents

Chapter 1

Introduction to Programming Language

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

Many programming languages have some form of written specification of their 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. Other languages, such as Perl, have a dominant implementation that is used as a reference.

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.

Definitions

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

- *Function and target:* A *computer programming language* is a language used to write computer programs, which involve a computer performing some kind of computation or algorithm and possibly control external devices such as printers, disk drives, robots, and so on. For example PostScript programs are frequently created by another program to control a computer printer or display. More generally, a programming language may describe computation on some, possibly abstract, machine. It is generally accepted that a complete specification for a programming language includes a description, possibly idealized, of a machine or processor for that language. In most practical contexts, a programming language involves a computer; consequently programming languages are usually defined and studied this way. Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

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

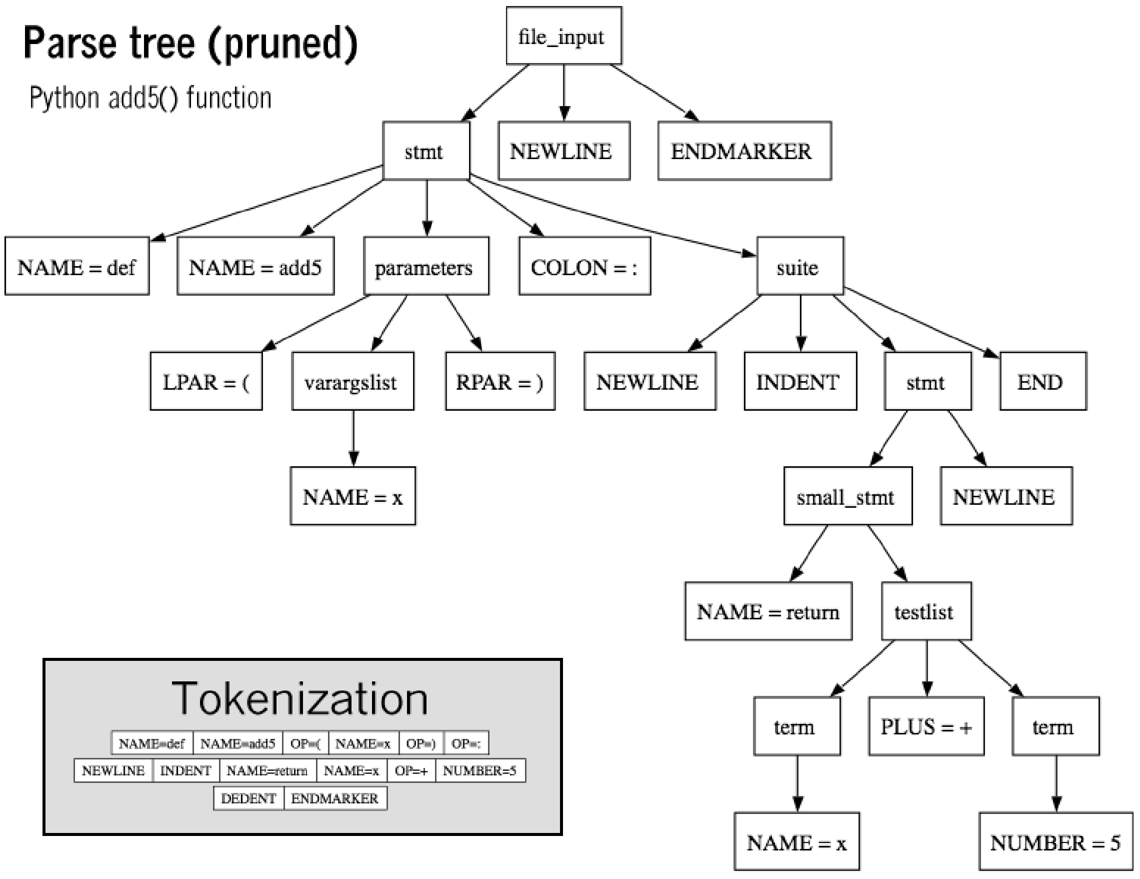
Markup languages like XML, HTML or troff, which define structured data, are not generally considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined. XSLT, for example, is a Turing complete XML dialect. Moreover,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

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

Elements

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

Syntax



Parse tree of Python code with inset tokenization

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

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

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

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

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

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

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

This grammar specifies the following:

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

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

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

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

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

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

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

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

The grammar needed to specify a programming language can be classified by its position in the Chomsky hierarchy. The syntax of most programming languages can be specified using a Type-2 grammar, i.e., they are context-free grammars. Some languages, including Perl and Lisp, contain constructs that allow execution during the parsing phase. Languages that have constructs that allow the programmer to alter the behavior of the parser make syntax analysis an undecidable problem, and generally blur the distinction between parsing and execution. In contrast to Lisp's macro system and Perl's `BEGIN` blocks, which may contain general computations, C macros are merely string replacements, and do not require code execution.

Static semantics

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

Typed versus untyped languages

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

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

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

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

Static versus dynamic typing

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

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

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

Weak and strong typing

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

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

An alternative definition for "weakly typed" refers to languages, such as Perl and JavaScript, which permit a large number of implicit type conversions. In JavaScript, for example, the expression `2 * x` implicitly converts `x` to a number, and this conversion succeeds even if `x` is `null`, `undefined`, an `Array`, or a string of letters. Such implicit conversions are often useful, but they can mask programming errors.

Strong and *static* are now generally considered orthogonal concepts, but usage in the literature differs. Some use the term *strongly typed* to mean *strongly, statically typed*, or, even more confusingly, to mean simply *statically typed*. Thus C has been called both strongly typed and weakly, statically typed.

Execution 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 *execution semantics* (also known as *dynamic 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.

Core library

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

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

Design and implementation

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

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

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

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

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

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

Specification

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

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

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

Implementation

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

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

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

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

Usage

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

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

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

Measuring language usage

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

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

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

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

Taxonomies

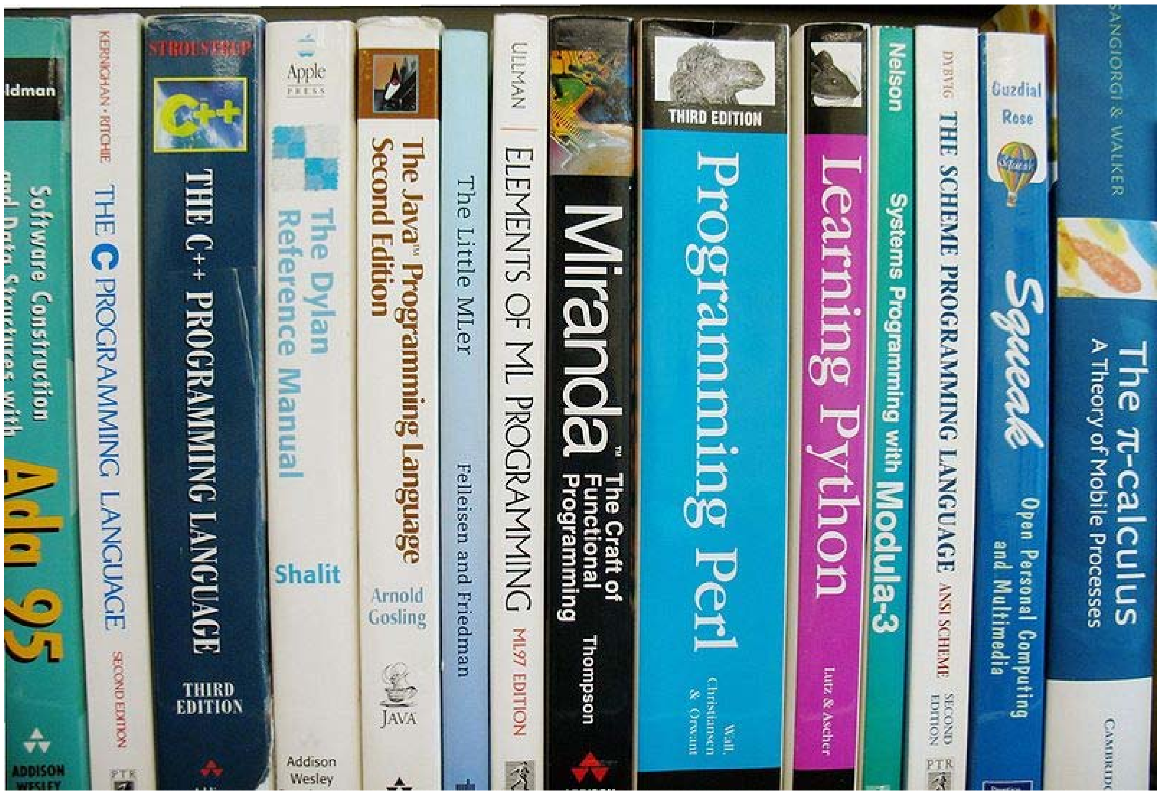
There is no overarching classification scheme for programming languages. A given programming language does not usually have a single ancestor language. Languages commonly arise by combining the elements of several predecessor languages with new ideas in circulation at the time. Ideas that originate in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

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

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

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

History


Software Construction and Data Structures with Ada 95
ADDISON WESLEY
KERNIGHAN • RITCHIE
THE C PROGRAMMING LANGUAGE
SECOND EDITION
PTR
STROUSTRUP
C++
THE C++ PROGRAMMING LANGUAGE
THIRD EDITION
Apple PRESS
The Dylan Reference Manual
Shalit
Addison Wesley
The Java™ Programming Language Second Edition
Arnold Gosling
JAVA
The Little MLer
Felleisen and Friedman
ULLMAN
ELEMENTS OF ML PROGRAMMING
ML97 EDITION
Miranda™ The Craft of Functional Programming
Thompson
THIRD EDITION
Programming Perl
Wall, Christiansen & Orwant
Learning Python
Lutz & Ascher
Nelson
Systems Programming with Modula-3
DYBVIG
THE SCHEME PROGRAMMING LANGUAGE
ANSI SCHEME
SECOND EDITION
PTR
Guzdial Rose
Squeak
Open Personal Computing and Multimedia
SANGIORGI & WALKER
The π-calculus
A Theory of Mobile Processes
CAMBRIDGE

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

Early developments

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

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

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

Refinement

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

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

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

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

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

Consolidation and growth

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

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

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

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

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

Chapter 2

Type System

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

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

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

Fundamentals

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

Major functions provided by type systems include:

- *Safety* – Use of types may allow a compiler to detect meaningless or probably invalid code. For example, we can identify an expression `3 / "Hello, World"` as invalid because the rules of arithmetic do not specify how to divide an integer by a string. As discussed below, strong typing offers more safety, but generally does not guarantee complete safety.

- *Optimization* – Static type-checking may provide useful compile-time information. For example, if a type requires that a value must align in memory at a multiple of 4 bytes, the compiler may be able to use more efficient machine instructions.
- *Documentation* – In more expressive type systems, types can serve as a form of documentation, since they can illustrate the intent of the programmer. For instance, timestamps may be represented as integers—but if a programmer declares a function as returning a timestamp type rather than merely an integer type, this documents part of the meaning of the function.
- *Abstraction* (or *modularity*) – Types allow programmers to think about programs at a higher level than the bit or byte, not bothering with low-level implementation. For example, programmers can think of a string as a collection of character values instead of as a mere array of bytes. Or, types can allow programmers to express the interface between two subsystems. This helps localize the definitions required for interoperability of the subsystems and prevents inconsistencies when those subsystems communicate.

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

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

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

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

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

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

Type checking

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

Static typing

A programming language is said to use static typing when type checking is performed during compile-time as opposed to run-time. Statically typed languages include Ada, ActionScript 3, C, C++, C#, Eiffel, F#, Go, JADE, Java, Fortran, Haskell, ML, Objective-C, Pascal, Perl (with respect to distinguishing scalars, arrays, hashes and subroutines) and Scala. Static typing is a limited form of program verification: accordingly, it allows many type errors to be caught early in the development cycle. Static type checkers evaluate only the type information that can be determined at compile time, but are able to verify that the checked conditions hold for all possible executions of the program, which eliminates the need to repeat type checks every time the program is executed. Program execution may also be made more efficient (i.e. faster or taking reduced memory) by omitting runtime type checks and enabling other optimizations.

Because they evaluate type information during compilation, and therefore lack type information that is only available at run-time, static type checkers are conservative. They will reject some programs that may be well-behaved at run-time, but that cannot be statically determined to be well-typed. For example, even if an expression `<complex test>` always evaluates to `true` at run-time, a program containing the code

```
      if <complex test> then 42 else <type error>
```

will be rejected as ill-typed, because a static analysis cannot determine that the `else` branch won't be taken. The conservative behaviour of static type checkers is advantageous when `<complex test>` evaluates to `false` infrequently: A static type checker can detect type errors in rarely used code paths. Without static type checking, even code coverage tests with 100% coverage may be unable to find such type errors. The tests may fail to detect such type errors because the combination of all places where values are created and all places where a certain value is used must be taken into account.

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

Dynamic typing

A programming language is said to be dynamically typed when the majority of its type checking is performed at run-time as opposed to at compile-time. In dynamic typing, values have types but variables do not; that is, a variable can refer to a value of any type. Dynamically typed languages include Erlang, Groovy, JavaScript, Lisp, Lua, Objective-C, Perl (with respect to user-defined types but not built-in types), PHP, Prolog, Python, Ruby, Smalltalk and Tcl. Compared to static typing, dynamic typing can be more flexible (e.g., by allowing programs to generate types and functionality based on run-time data), though at the expense of fewer *a priori* guarantees. This is because a dynamically typed language accepts and attempts to execute some programs which may be ruled as invalid by a static type checker. The term "dynamic language" means something different ("runtime dynamism") and a dynamic language is not necessarily dynamically typed.

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. This operation 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 these 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 conversely cannot search as comprehensively for the errors that static type checking is able to detect.

Combinations of dynamic and static typing

The presence of static typing in a programming language does not necessarily imply the absence of all dynamic typing mechanisms. For example, Java, and various other object-oriented languages, while using static typing, require for certain operations (downcasting) the support of runtime type tests, a form of dynamic typing.

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

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

Static and dynamic type checking in practice

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

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

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

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

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

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

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

Dynamic typing typically makes metaprogramming more effective and easier to use. For example, C++ templates are typically more cumbersome to write than the equivalent Ruby or Python code. More advanced run-time constructs such as metaclasses and introspection are often more difficult to use in statically typed languages.

Strong and weak typing

One definition of *strongly typed* involves preventing success for an operation on arguments which have the wrong type. A C cast gone wrong exemplifies the problem of absent strong typing; 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.

Weak typing means that a language implicitly converts (or casts) types when used. For example, we may have:

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

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

Safely and unsafely typed systems

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

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

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

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

Now let us look at the same example in C:

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

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

Polymorphism and types

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

Duck typing

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

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

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

Initially coined by Alex Martelli in the Python community, 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".

Specialized type systems

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

Dependent types

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

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

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

Linear types

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

Intersection types

Intersection types are types describing values that belong to *both* of two other given types with overlapping value sets. For example, in most implementations of C the signed char has range -128 to 127 and the unsigned char has range 0 to 255, so the intersection type of these two types would have range 0 to 127. Such an intersection type could be safely passed into functions expecting *either* signed or unsigned chars, because it is compatible with both types.

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

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

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

Union types

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

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

Existential types

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

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

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

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

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

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

Explicit or implicit declaration and inference

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

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

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

Chapter 3

Ada (programming language) and ALGOL

Ada (programming language)

Ada	
Paradigm	Multi-paradigm
Appeared in	1983
Designed by	Ada1983: Jean Ichbiah; Ada1995: Tucker Taft; Ada2005: Tucker Taft
Stable release	Ada 2005 (2007)
Typing discipline	static, strong, safe, nominative
Major implementations	GNAT
Dialects	Ada 83, Ada 95, Ada 2005
Influenced by	ALGOL 68, Pascal, C++ (Ada 95), Smalltalk (Ada 95), Java (Ada 2005)
Influenced	C++, Eiffel, PL/SQL, VHDL, Ruby, Java

Ada is a structured, statically typed, imperative, wide-spectrum, and object-oriented high-level computer programming language, extended from Pascal and other languages. It 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 is strongly typed and compilers are validated for reliability in mission-critical applications, such as avionics software. Ada is an international standard; the current version (known as Ada

2005) is defined by joint ISO/ANSI standard, combined with major Amendment ISO/IEC 8652:1995/Amd 1:2007.

Ada was named after Ada Lovelace (1815–1852), who is often credited as being the first computer programmer.

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), exception handling, and generics. Ada 95 added support for object-oriented programming, including dynamic dispatch.

Ada supports run-time checks to protect against access to unallocated memory, buffer overflow errors, 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 or injury. Examples of systems where Ada is used include avionics, weapon systems (including thermonuclear weapons), and satellites.

Ada also supports a large number of compile-time checks 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. Ada is designed to detect small problems in very large software systems. For example, Ada detects each misspelled variable (due to the rule to declare each variable name), and Ada pinpoints unclosed if-statements, which require "END IF" rather than mismatching with any "END" token. Also, Ada can spot procedure calls with incorrect parameters, which is a common problem in large, complex software where most of the statements are procedure calls.

Ada's dynamic memory management is high-level and type-explicit, requiring explicit instantiation of the Unchecked_Deallocation package to explicitly free allocated memory. The specification does not require any particular implementation. Though the semantics of the language allow automatic garbage collection of inaccessible objects, most implementations do not support it. Ada does support a limited form of region-based storage management. Invalid accesses can always be detected at run time (unless of course the check is turned off) and sometimes at compile time.

The syntax of Ada is simple, consistent and readable. It minimizes choices of ways to perform basic operations, and prefers English keywords (e.g. "or else") to symbols (e.g. "||"). 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". Conditional statements are closed with "end if", avoiding a *dangling else* that could pair with the wrong nested if-expression in other languages.

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. It can be argued that 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, 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 working group created a series of language requirements documents—the Strawman, Woodenman, Tinman, Ironman and Steelman documents. Many existing languages were formally reviewed, but the team concluded in 1977 that no existing language met the specifications.

Requests for proposals for a new programming language were issued and four contractors were hired to develop their proposals under the names of Red (Intermetrics led by Benjamin Brosgol), Green (CII Honeywell Bull, led by Jean Ichbiah), Blue (SofTech, led by John Goodenough), and Yellow (SRI International, led by Jay Spitzen). In April 1978, after public scrutiny, the Red and Green proposals passed to the next phase. In May 1979, the Green proposal, designed by Jean Ichbiah at CII Honeywell Bull, was chosen and given the name Ada—after Augusta Ada, Countess of Lovelace. This proposal was influenced by the 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, but subsequently seemed to recant in the foreword he wrote for an Ada textbook.

Ada attracted much attention from the programming community as a whole during its early days. Its backers and others predicted that it might become a dominant language for general purpose programming and not just defense-related work. Ichbiah publicly stated that within ten years, only two programming languages would remain, Ada and Lisp. Early Ada compilers struggled to implement the large, complex language, and both compile-time and run-time performance tended to be slow and tools primitive. Compiler vendors expended most of their efforts in passing the massive, language-conformance-testing, government-required "ACVC" validation suite that was required in another novel feature of the Ada language effort.

Augusta Ada King, Countess of Lovelace

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.

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 Ada is a strongly typed language and has other safety-critical support features, it has found a niche outside the military in commercial aviation projects, where a software bug can cause fatalities. 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 (for the time) distributed processing, a distributed Ada database, and object-oriented design. It is also used to program the processors of the French TVM in-cab signalling system on the LGVs.

Standardization

The language became an ANSI standard in 1983 (ANSI/MIL-STD 1815A), 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) 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, the current version of the standard, was published on March 9, 2007. Work on the next significant Ada Amendment is planned to be completed by 2012.(ISO/IEC 8652:201z Ed. 3)

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 Algol60, like 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:

```
with Ada.Text_IO;

procedure Hello is
begin
  Ada.Text_IO.Put_Line("Hello, world!");
```

```
end Hello;
```

Ada also provides alternative constructions that are more streamlined.

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.

For example a date might be represented as:

```
type Day   is range    1 ..   31;
type Month is range    1 ..   12;
type Year  is range 1800 .. 2100;

type Date is
   record
     Day   : Day;
     Month : Month;
     Year  : Year;
   end record;
```

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");
end case;
```

Packages, procedures and functions

Ada programs consist of packages, procedures and functions.

```
with Ada.Text_IO;

package Mine is

  type Integer is range 1 .. 11;

  i : Integer := Integer'First;

  procedure Print (j: in out Integer) is

    function Next (k: in Integer) return Integer is
    begin
      return k + 1;
    end Next;

  begin
    Ada.Text_IO.Put_Line ('The total is: ', j);
    j := Next (j);
  end Print;

begin
  while i < Integer'Last loop
    Print (i);
  end loop;
end Mine;
```

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 goto labels, constants, types, variables, and other procedures, functions and packages, which can be declared in any order.

ALGOL

ALGOL	
Paradigm	procedural, imperative, structured
Appeared in	1958

Designed by	Bauer, Bottenbruch, Rutishauser, Samelson, Backus, Katz, Perlis, Wegstein, Naur, Vauquois, van Wijngaarden, Woodger, Green, McCarthy
Major implementations	MARST
Influenced	Most subsequent imperative languages (so-called *ALGOL-like* languages) e.g. Simula, C, CPL, Pascal, Ada

ALGOL (short for **ALGO**rithmic **L**anguage) is a family of imperative computer programming languages originally developed in the mid 1950s which greatly influenced many other languages and became the *de facto* way algorithms were described in textbooks and academic works for almost the next 30 years. It was designed to avoid some of the perceived problems with FORTRAN and eventually gave rise to many other programming languages, including BCPL, B, Pascal, Simula, C, and many others. ALGOL introduced code blocks and the `begin` and `end` pairs for delimiting them and it was also the first language implementing nested function definitions with lexical scope. Fragments of ALGOL-like syntax are sometimes still used as pseudocode (notations used to describe algorithms for human readers).

There were three major specifications:

- ALGOL 58 - originally proposed to be called **IAL** (for **I**nternational **A**lgorithmic **L**anguage).
- ALGOL 60 - first implemented as *X1 ALGOL 60* in mid 1960 - revised 1963
- ALGOL 68 - revised 1973 - introduced new elements including flexible arrays, slices, parallelism, operator identification, and various extensibility features.

Niklaus Wirth based his own ALGOL W on ALGOL 60 before developing Pascal. Algol-W was intended to be the next generation ALGOL but the ALGOL 68 committee decided on a design that was more complex and advanced rather than a cleaned simplified ALGOL 60. The official ALGOL versions are named after the year they were first published.

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

Important implementations

The International Algorithmic Language (IAL) was extremely influential and generally considered the ancestor of most of the modern programming languages (the so-called Algol-like languages). Additionally, in computer science, **ALGOL object code** was a simple and compact and stack-based instruction set architecture mainly used in teaching compiler construction and other high order language (of which Algol is generally considered the first).

History

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

ALGOL was used mostly by research computer scientists in the United States and in Europe. Its use in commercial applications was hindered by the absence of standard input/output facilities in its description and the lack of interest in the language by large computer vendors. ALGOL 60 did however become the standard for the publication of algorithms and had a profound effect on future language development.

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

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

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

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

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

Both John Backus and Peter Naur served on the committee which created ALGOL 60 as did Wally Feurzeig, who later created Logo.

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

Algol and programming language research

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

IAL implementations timeline

To date there have been at least 70 augmentations, extensions, derivations and sublanguages of Algol 60.

Name	Year	Author	State	Description	Target CPU
ZMMD-implementation	1958	Friedrich L. Bauer, Heinz Rutishauser, Klaus Samelson, Hermann Bottenbruch	Germany	implementation of ALGOL 58	Z22 (later Zuse's Z23 was delivered with an Algol 60 compiler)
X1 ALGOL 60	Summer 1960	Edsger W. Dijkstra and Jaap A. Zonneveld	Netherlands	First implementation of ALGOL 60	Electrologica X1
Elliott ALGOL	1960s	C. A. R. Hoare	UK	Subject of the famous Turing lecture	Elliott 803 & the Elliott 503
JOVIAL	1960	Jules Schwarz	USA	Was the DOD HOL prior to Ada (programming language)	Various
Burroughs Algol (Several variants)	1961	Burroughs Corporation (with participation by Hoare, Dijkstra, and	USA	Basis of the Burroughs (and now Unisys MCP based)	Burroughs large systems and their midrange as

		others)		computers	well.
Case ALGOL	1961		USA	Simula was originally contracted as a simulation extension of the Case ALGOL	UNIVAC 1107
GOGOL	1961	Bill McKeeman	USA	For ODIN time-sharing system	PDP-1
X1 Algol 60	1961	Edsger W. Dijkstra and J.A. Zonneveld	Netherlands	Mathematical Centre, Amsterdam	X1
RegneCentralen ALGOL	1961	Peter Naur, Jørn Jensen	Denmark	Inplementation of full Algol 60	DASK at Regnecentralen
Dartmouth ALGOL 30	1962	Thomas Eugene Kurtz et al.	USA		LGP-30
USS 90 Algol	1962	L. Petrone	Italy		
Algol Translator	1962	G. van der Mey and W.L. van der Poel	Netherlands	Staatsbedrijf der Posterijen, Telegrafie en Telefonie	ZEBRA
Kidsgrove Algol	1963	F. G. Duncan	UK		English Electric Company KDF9
VALGOL	1963	Val Schorre	USA	A test of the META II compiler compiler	
Whetstone	1964	Brian Randell and L J Russell	UK	Atomic Power Division of English Electric Company. Precursor to Ferranti Pegasus, National Physical Laboratories ACE and English Electric DEUCE implementations.	English Electric Company KDF9
NU ALGOL	1965		Norway		UNIVAC
ALGEK	1965		USSR	Minsk-22	АЛГЭК, based

					on ALGOL-60 and COBOL support, for economical tasks
MALGOL	1966	publ. A. Viil, M Kotli & M. Rakhendi,	Estonian SSR	Minsk-22	
ALGAMS	1967	GAMS group (ГАМС, группа автоматизации программирования для машин среднего класса), cooperation of Comecon Academies of Science	Comecon	Minsk-22, later ES EVM, BESM	
ALGOL/ZAM	1967		Poland		Polish ZAM computer
Simula 67	1967	Ole-Johan Dahl and Kristen Nygaard	Norway	Algol 60 with classes	UNIVAC 1107
Chinese Algol	1972		China	Chinese characters, expressed via the Symbol system	
DG/L	1972		USA		DG Eclipse family of Computers
S-algol	1979	Prof. Ron Morrison	UK	Addition of orthogonal datatypes with intended use as a teaching language	PDP-11 with a subsequent implementation on the Java VM

The Burroughs dialects included special Bootstrapping dialects such as ESPOL and NEWP.

Properties

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

ALGOL 60 allowed for two evaluation strategies for parameter passing: the common call-by-value, and call-by-name. Call-by-name had certain limitations in contrast to call-by-reference, making it an undesirable feature in imperative language design. For example, it is impossible in ALGOL 60 to develop a procedure that will swap the values of two parameters if the actual parameters that are passed in are an integer variable and an array that is indexed by that same integer variable. Think of passing a pointer to swap(i, A[i]) in to a function. Now that every time swap() is referenced, it's reevaluated. Say i := 1 and A[i] := 2, so every time swap() is referenced it'll return the other combination of the values ([1,2], [2,1], [1,2] and so on). Another problematic situation is passing a random() function.

However, call-by-name is still beloved of ALGOL implementors for the interesting "thunks" that are used to implement it. Donald Knuth devised the "man or boy test" to separate compilers that correctly implemented "recursion and non-local references." This test contains an example of call-by-name.

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

Examples and portability issues

Code sample comparisons

ALGOL 60

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

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

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

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

READ D'

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

PUNCH(3) sends output to the teleprinter rather than the tape punch.
SAMELINE suppresses the carriage return + line feed normally printed between arguments.
ALIGNED(1,6) controls the format of the output with 1 digit before and 6 after the decimal point.

ALGOL 68

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

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

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

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

```
floating point algol68 test:
(
  real a,b,c,d;
  printf(($pg$,"Enter d:"));
  read(d);

  for step from 0 while a:=step*d; a <= 2*pi do
    printf($l$);
    b := sin(a);
    c := cos(a);
    printf(($z-d.6d$,a,b,c))
  od
)
```

printf - sends output to the **file** *stand out*.
printf(p); - selects a *new page*.
printf(l); - selects a *new line*.
printf(($z-d.6d$,a,b,c)) - formats output with 1 digit before and 6 after the decimal point.

Timeline: Hello world

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

ALGOL 58 (IAL)

ALGOL 58 had no I/O facilities.

ALGOL 60 family

Since ALGOL 60 had no I/O facilities, there is no portable hello world program in ALGOL. The following program could (and still will) compile and run on an ALGOL implementation for a Unisys A-Series mainframe, and is a straightforward simplification of code taken from The Language Guide at the University of Michigan-Dearborn Computer and Information Science Department Hello world! ALGOL Example Program page.

```
BEGIN
  FILE F(KIND=REMOTE);
  EBCDIC ARRAY E[0:11];
  REPLACE E BY "HELLO WORLD!";
  WRITE(F, *, E);
END.
```

A simpler program using an inline format:

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

An even simpler program using the Display statement:

```
BEGIN DISPLAY("HELLO WORLD!") END.
```

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

```
 program HiFolks;
 begin
    print ‘Hello world’;
 end;
```

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

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

The ICT 1900 Algol I/O version allowed input from paper tape or punched card. Paper tape 'full' mode allowed lower case. Output was to a line printer.

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

ALGOL 68

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

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

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

Timeline of ALGOL special characters

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

1960: IFIP - The Algol 60 language and report included several mathematical symbols which are available on modern computers and operating systems, but, unfortunately, were not supported on most computing systems at the time.

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

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

1968: The "Algol 68 Report" - used existing ALGOL characters, and further adopted →, ↑, □, ⌊, ⌈, ○, ⊥ and ¢ characters which can be found on the IBM 2741 keyboard with "golf-ball" print heads inserted (such as the APL golfball), these became available in the mid 1960s while ALGOL 68 was being drafted. The report was translated into Russian, German, French and Bulgarian, and allowed programming in languages with larger character sets, e.g. Cyrillic alphabet of the Russian BESM-4. All ALGOL's characters are also part of the unicode standard and most of them are available in several popular fonts.

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

Chapter 4

Erlang (Programming Language) and Forth (Programming Language)

Erlang (programming language)

Erlang

Paradigm	multi-paradigm: concurrent, functional
Appeared in	1986
Designed by	Ericsson
Developer	Ericsson
Stable release	R14B (September 15, 2010; 2 months ago)

Typing discipline	dynamic, strong
Major implementations	Erlang
Influenced by	Prolog
Influenced	Clojure, Scala
License	Modified MPL

Erlang 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. For concurrency it follows the Actor model. It was designed by Ericsson to support distributed, fault-tolerant, soft-real-time, non-stop applications. The first version was developed by Joe Armstrong in 1986. It supports hot swapping, thus code can be changed without stopping a system. It was originally a proprietary language within Ericsson, but was released as open source in 1998.

While threads are considered a complicated and error-prone topic 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.

History

The name "Erlang", attributed to Bjarne Däcker, has been understood either as a reference to Danish mathematician and engineer Agner Krarup Erlang, or alternatively, as an abbreviation of "Ericsson Language".

Erlang was designed with the aim of improving the development of telephony applications. The initial version of Erlang was implemented in Prolog.

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. 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. The implementation was open sourced at the end of the year. The ban at Ericsson was eventually lifted, and Armstrong was re-hired by Ericsson in 2004.

In 2006, native symmetric multiprocessing support was added to the runtime system and virtual machine.

Philosophy

The philosophy used to develop Erlang fits equally well with the development of Erlang-based systems. 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 language

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) -> 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) when is_list(A), is_list(B) -> length(A) <
length(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) when is_list(A), is_list(B) -> length(A) < length(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.

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 obtain 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 are represented as IEEE 754 64-bit floating point numbers. Real numbers in the range $\pm10^{308}$ can be represented by an Erlang float.
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 message 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 `[Dh|Dt]` denotes a list whose first element is `Dh`, and whose remaining elements are the list `Dt`. The syntax `[]` denotes an empty list. The syntax `[D1,D2,..,Dn]` is short for `[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]`.
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.

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 somewhat similar to Java's original “green threads”. Like operating system processes (and unlike green threads and operating system threads) they have no shared state between them. The estimated minimal overhead for each is 300 words, thus many of them can be created without degrading performance: a benchmark with 20 million processes has been successfully performed. 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. 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. It also supports interpreting, directly from source code via abstract tree, via script as of R11B-5.

Hot code loading and modules

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. 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.

Erlang is available for many Unix-like operating systems, including Mac OS X, and for Microsoft Windows.

Projects using Erlang

Projects using Erlang include:

- CouchDB, a document based database that uses MapReduce
- ejabberd, an Extensible Messaging and Presence Protocol (XMPP) instant messaging server
    - Facebook chat system, based on ejabberd
- Erlyvideo, high efficiency multiprotocol videostreaming server
- GitHub egitd, a replacement for stock git-daemon that ships with Git
- Issuu, an online digital publisher
- Membase, database management system optimized for storing data behind interactive web applications.
- RabbitMQ, an implementation of Advanced Message Queuing Protocol (AMQP)
- SimpleDB, a distributed database that is part of Amazon Web Services
- Twitterfall, a service to view trends and patterns from Twitter
- Wings 3D, a 3D modeller
- Yaws web server

Clones

Erlang has inspired clones of its concurrency facilities for other languages:

- Reia
- Scala

Forth (programming language)

	Forth
Paradigm	Procedural, stack-oriented, reflective
Appeared in	1970s
Designed by	Charles H. Moore
Typing discipline	typeless
Major implementations	Forth, Inc., Gforth, MPE
Dialects	colorForth, MUF, Open Firmware

Influenced by	Burroughs large systems, Lisp, APL
Influenced	Factor, Cat, PostScript, RPL, REBOL

Forth is a structured, imperative, reflective, extensible, stack-based computer programming language and programming environment. Although not an acronym, the language's name is sometimes spelled with all capital letters as FORTH, following the customary usage during its earlier years.

A procedural programming language without type checking, Forth features both interactive execution of commands (making it suitable as a shell for systems that lack a more formal operating system) and the ability to compile sequences of commands for later execution. Some Forth implementations (usually early versions or those written to be extremely portable) compile threaded code, but many implementations today generate optimized machine code like other language compilers.

Although not as popular as other programming systems, Forth has enough support to keep several language vendors and contractors in business. Forth is currently used in boot loaders such as Open Firmware, space applications, and other embedded systems. Gforth, an implementation of Forth by the GNU Project is actively maintained, the last release in December 2008. The 1994 standard is currently undergoing revision, provisionally titled Forth 200x.

Overview

A Forth environment combines the compiler with an interactive shell. The user interactively defines and runs subroutines, or "words," in a virtual machine similar to the runtime environment. Words can be tested, redefined, and debugged as the source is entered without recompiling or restarting the whole program. All syntactic elements, including variables and basic operators, appear as such procedures. Even if a particular word is optimized so as not to require a subroutine call, it is also still available as a subroutine. On the other hand, the shell may compile interactively typed commands into machine code before running them. (This behavior is common, but not required.) Forth environments vary in how the resulting program is stored, but ideally running the program has the same effect as manually re-entering the source. This contrasts with the combination of C with Unix shells, wherein compiled functions are a special class of program objects and interactive commands are strictly interpreted. Most of Forth's unique properties result from this principle. By including interaction, scripting, and compilation, Forth was popular on computers with limited resources, such as the BBC Micro and Apple II series, and remains so in applications such as firmware and small microcontrollers. Where C compilers may now generate code with more compactness and performance, Forth retains the advantage of interactivity.

Stacks

Most programming environments with recursive subroutines use a stack for control flow. This structure typically also stores local variables, including subroutine parameters (in a call by value system such as C). Forth often does not have local variables, however, nor is it call-by-value. Instead, intermediate values are kept in a second stack. Words operate directly on the topmost values in the first stack. It may therefore be called the "parameter" or "data" stack, but most often simply "the" stack. The second, function-call stack is then called the "linkage" or "return" stack, abbreviated *rstack*. Special rstack manipulation functions provided by the kernel allow it to be used for temporary storage within a word, but otherwise it cannot be used to pass parameters or manipulate data.

Most words are specified in terms of their effect on the stack. Typically, parameters are placed on the top of the stack before the word executes. After execution, the parameters have been erased and replaced with any return values. For arithmetic operators, this follows the rule of reverse Polish notation.

Maintenance

Forth is a simple yet extensible language; its modularity and extensibility permit the writing of high-level programs such as CAD systems. However, extensibility also helps poor programmers to write incomprehensible code, which has given Forth a reputation as a "write-only language". Forth has been used successfully in large, complex projects, while applications developed by competent, disciplined professionals have proven to be easily maintained on evolving hardware platforms over decades of use. Forth has a niche both in astronomical and space applications. Forth is still used today in many embedded systems (small computerized devices) because of its portability, efficient memory use, short development time, and fast execution speed. It has been implemented efficiently on modern RISC processors, and processors that use Forth as machine language have been produced. Other uses of Forth include the Open Firmware boot ROMs used by Apple, IBM, Sun, and OLPC XO-1; and the FICL-based first stage boot controller of the FreeBSD operating system.

History

Forth evolved from Charles H. Moore's personal programming system, which had been in continuous development since 1958. Forth was first exposed to other programmers in the early 1970s, starting with Elizabeth Rather at the US National Radio Astronomy Observatory. After their work at NRAO, Charles Moore and Elizabeth Rather formed FORTH, Inc. in 1973, refining and porting Forth systems to dozens of other platforms in the next decade.

Forth is so named because in 1968 "[t]he file holding the interpreter was labeled FOURTH, for 4th (next) generation software—but the IBM 1130 operating system restricted file names to 5 characters." Moore saw Forth as a successor to compile-link-go third-generation programming languages, or software for "fourth generation" hardware, not a fourth-generation programming language as the term has come to be used.

Because Charles Moore had frequently moved from job to job over his career, an early pressure on the developing language was ease of porting to different computer architectures. A Forth system has often been used to bring up new hardware. For example, Forth was the first resident software on the new Intel 8086 chip in 1978 and MacFORTH was the first resident development system for the first Apple Macintosh in 1984.

FORTH, Inc.'s microFORTH was developed for the Intel 8080, Motorola 6800, and Zilog Z80 microprocessors starting in 1976. MicroFORTH was later used by hobbyists to generate Forth systems for other architectures, such as the 6502 in 1978. Wide dissemination finally led to standardization of the language. Common practice was codified in the de facto standards FORTH-79 and FORTH-83 in the years 1979 and 1983, respectively. These standards were unified by ANSI in 1994, commonly referred to as ANS Forth.

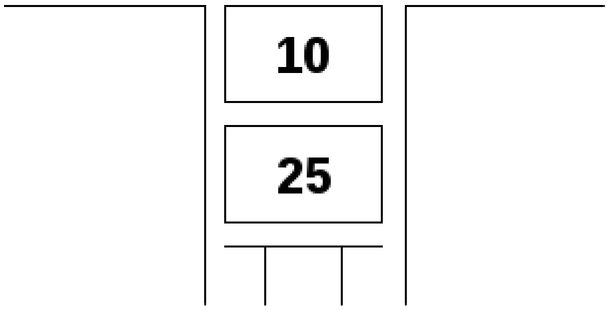
Forth became very popular in the 1980s because it was well suited to the small microcomputers of that time, as it is compact and portable. At least one home computer, the British Jupiter ACE, had Forth in its ROM-resident operating system. The Canon Cat also used Forth for its system programming. Rockwell also produced single-chip microcomputers with resident Forth kernels, the R65F11 and R65F12. A complete family tree at TU-Wien.

Programmer's perspective

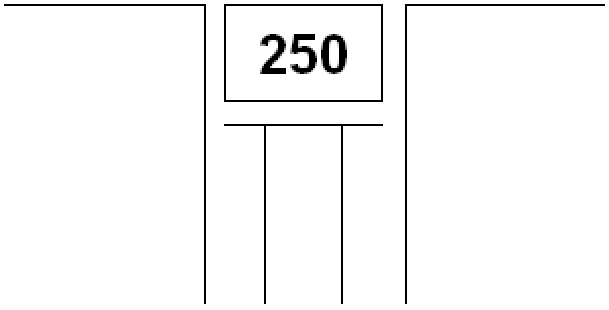
Forth relies heavily on explicit use of a data stack and reverse Polish notation (RPN or postfix notation), commonly used in calculators from Hewlett-Packard. In RPN, the operator is placed after its operands, as opposed to the more common infix notation where the operator is placed between its operands. Postfix notation makes the language easier to parse and extend; Forth does not use a BNF grammar, and does not have a monolithic compiler. Extending the compiler only requires writing a new word, instead of modifying a grammar and changing the underlying implementation.

Using RPN, one could get the result of the mathematical expression `(25 * 10 + 50)` this way:

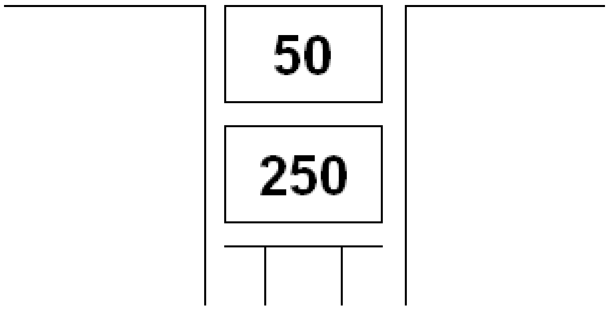
```
25 10 * 50 + . <cr> 300 ok
```



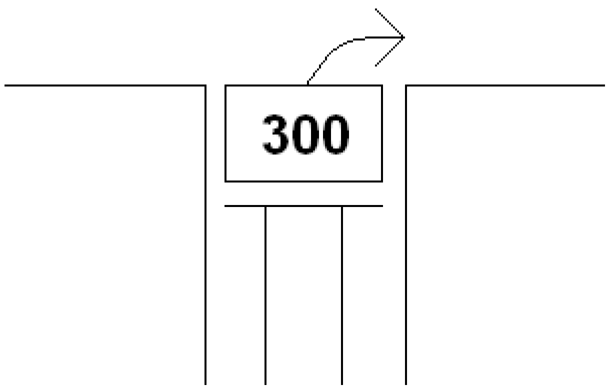
This command line first puts the numbers 25 and 10 on the implied stack.



The word * multiplies the two numbers on the top of the stack and replaces them with their product.



Then the number 50 is placed on the stack.



The word + adds it to the previous product. Finally, the . command prints the result to the user's terminal.

Even Forth's structural features are stack-based. For example:

```
: FLOOR5 ( n -- n' )   DUP 6 < IF DROP 5 ELSE 1 - THEN ;
```

This code defines a new word (again, *word* is the term used for a subroutine) called `FLOOR5` using the following commands: `DUP` duplicates the number on the stack; `6` places a 6 on top of the stack; `<` compares the top two numbers on the stack (6 and the `DUP`ed input), and replaces them with a true-or-false value; `IF` takes a true-or-false value and chooses to execute commands immediately after it or to skip to the `ELSE`; `DROP` discards the value on the stack; and `THEN` ends the conditional. The text in parentheses is a comment, advising that this word expects a number on the stack and will return a possibly changed number. The `FLOOR5` word is equivalent to this function written in the C programming language using ternary operator:

```
int floor5(int v) { return (v < 6) ? 5 : (v - 1); }
```

This function is written more succinctly as:

```
: FLOOR5 ( n -- n' ) 1- 5 MAX ;
```

You would run this word as follows:

```
1 FLOOR5 . <cr> 5 ok
8 FLOOR5 . <cr> 7 ok
```

First the interpreter pushes a number (1 or 8) onto the stack, then it calls FLOOR5, which pops off this number again and pushes the result. Finally, a call to "." pops the result and prints it to the user's terminal.

Facilities

Forth parsing is simple, despite having no explicit grammar. The interpreter reads a line of input from the user input device, which is then parsed for a word using spaces as a delimiter; some systems recognise additional whitespace characters. When the interpreter finds a word, it tries to look the word up in the *dictionary*. If the word is found, the interpreter executes the code associated with the word, and then returns to parse the rest of the input stream. If the word isn't found, the word is assumed to be a number, and an attempt is made to convert it into a number and push it on the stack; if successful, the interpreter continues parsing the input stream. Otherwise, if both the lookup and number conversion fails, the interpreter prints the word followed by an error message indicating the word is not recognised, flushes the input stream, and waits for new user input.

The definition of a new word is started with the word `:` (colon) and ends with the word `;` (semi-colon). For example

```
: X DUP 1+ . . ;
```

will compile the word `X`, and makes the name findable in the dictionary. When executed by typing `10 X` at the console this will print `11 10`.

Most Forth systems include a specialized assembler that produces executable words. The assembler is a special dialect of the compiler. Forth assemblers often use a reverse-polish syntax in which the parameters of an instruction precede the instruction. The usual design of a Forth assembler is to construct the instruction on the stack, then copy it into memory as the last step. Registers may be referenced by the name used by the manufacturer, numbered (0..n, as used in the actual operation code) or named for their purpose in the Forth system: e.g. "S" for the register used as a stack pointer.

Operating system, files, and multitasking

Classic Forth systems traditionally use neither operating system nor file system. Instead of storing code in files, source-code is stored in disk blocks written to physical disk addresses. The word `BLOCK` is employed to translate the number of a 1K-sized block of disk space into the address of a buffer containing the data, which is managed automatically by the Forth system. Some implement contiguous disk files using the system's disk access, where the files are located at fixed disk block ranges. Usually these are implemented as fixed-length binary records, with an integer number of records per disk block. Quick searching is achieved by hashed access on key data.

Multitasking, most commonly cooperative round-robin scheduling, is normally available (although multitasking words and support are not covered by the ANSI Forth Standard). The word `PAUSE` is used to save the current task's execution context, to locate the next task, and restore its execution context. Each task has its own stacks, private copies of some control variables and a scratch area. Swapping tasks is simple and efficient; as a result, Forth multitaskers are available even on very simple microcontrollers such as the Intel 8051, Atmel AVR, and TI MSP430.

By contrast, some Forth systems run under a host operating system such as Microsoft Windows, Linux or a version of Unix and use the host operating system's file system for source and data files; the ANSI Forth Standard describes the words used for I/O. Other non-standard facilities include a mechanism for issuing calls to the host OS or windowing systems, and many provide extensions that employ the scheduling provided by the operating system. Typically they have a larger and different set of words from the stand-alone Forth's `PAUSE` word for task creation, suspension, destruction and modification of priority.

Self-compilation and cross compilation

A fully featured Forth system with all source code will compile itself, a technique commonly called meta-compilation by Forth programmers (although the term doesn't exactly match meta-compilation as it is normally defined). The usual method is to redefine the handful of words that place compiled bits into memory. The compiler's words use specially-named versions of fetch and store that can be redirected to a buffer area in memory. The buffer area simulates or accesses a memory area beginning at a different address than the code buffer. Such compilers define words to access both the target computer's memory, and the host (compiling) computer's memory.

After the fetch and store operations are redefined for the code space, the compiler, assembler, etc. are recompiled using the new definitions of fetch and store. This effectively reuses all the code of the compiler and interpreter. Then, the Forth system's code is compiled, but this version is stored in the buffer. The buffer in memory is written to disk, and ways are provided to load it temporarily into memory for testing. When the new version appears to work, it is written over the previous version.

There are numerous variations of such compilers for different environments. For embedded systems, the code may instead be written to another computer, a technique known as cross compilation, over a serial port or even a single TTL bit, while keeping the word names and other non-executing parts of the dictionary in the original compiling computer. The minimum definitions for such a forth compiler are the words that fetch and store a byte, and the word that commands a Forth word to be executed. Often the most time-consuming part of writing a remote port is constructing the initial program to implement fetch, store and execute, but many modern microprocessors have integrated debugging features (such as the Motorola CPU32) that eliminate this task.

Structure of the language

The basic data structure of Forth is the "dictionary" which maps "words" to executable code or named data structures. The dictionary is laid out in memory as a tree of linked lists with the links proceeding from the latest (most recently) defined word to the oldest, until a sentinel, usually a NULL pointer, is found. A context switch causes a list search to start at a different leaf. A linked list search continues as the branch merges into the main trunk leading eventually back to the sentinel, the root. There can be several dictionaries. In rare cases such as meta-compilation a dictionary might be isolated and stand-alone. The effect resembles that of nesting namespaces and can overload keywords depending on the context.

A defined word generally consists of *head* and *body* with the head consisting of the *name field* (NF) and the *link field* (LF) and body consisting of the *code field* (CF) and the *parameter field* (PF).

Head and body of a dictionary entry are treated separately because they may not be contiguous. For example, when a Forth program is recompiled for a new platform, the head may remain on the compiling computer, while the body goes to the new platform. In some environments (such as embedded systems) the heads occupy memory unnecessarily. However, some cross-compilers may put heads in the target if the target itself is expected to support an interactive Forth.

Dictionary entry

The exact format of a dictionary entry is not prescribed, and implementations vary. However, certain components are almost always present, though the exact size and order may vary. Described as a structure, a dictionary entry might look this way:

```
structure
  byte:       flag           \ 3bit flags + length of word's name
  char-array: name           \ name's runtime length isn't known at
compile time
  address:    previous       \ link field, backward ptr to previous
word
  address:    codeword       \ ptr to the code to execute this word
  any-array:  parameterfield \ unknown length of data, words, or
opcodes
end-structure forthword
```

The name field starts with a prefix giving the length of the word's name (typically up to 32 bytes), and several bits for flags. The character representation of the word's name then follows the prefix. Depending on the particular implementation of Forth, there may be one or more NUL ('\0') bytes for alignment.

The link field contains a pointer to the previously defined word. The pointer may be a relative displacement or an absolute address that points to the next oldest sibling.

The code field pointer will be either the address of the word which will execute the code or data in the parameter field or the beginning of machine code that the processor will execute directly. For colon defined words, the code field pointer points to the word that will save the current Forth instruction pointer (IP) on the return stack, and load the IP with the new address from which to continue execution of words. This is the same as what a processor's call/return instructions does.

Structure of the compiler

The compiler itself is not a monolithic program. It consists of Forth words visible to the system, and usable by a programmer. This allows a programmer to change the compiler's words for special purposes.

The "compile time" flag in the name field is set for words with "compile time" behavior. Most simple words execute the same code whether they are typed on a command line, or embedded in code. When compiling these, the compiler simply places code or a threaded pointer to the word.

The classic examples of compile-time words are the control structures such as `IF` and `WHILE`. All of Forth's control structures, and almost all of its compiler are implemented as compile-time words. All of Forth's control flow words are executed during compilation to compile various combinations of the primitive words `BRANCH` and `?BRANCH` (branch if false). During compilation, the data stack is used to support control structure balancing, nesting, and backpatching of branch addresses. The snippet:

```
... DUP 6 < IF DROP 5 ELSE 1 - THEN ...
```

would be compiled to the following sequence inside of a definition:

```
... DUP LIT 6 < ?BRANCH 5  DROP LIT 5  BRANCH 3  LIT 1 - ...
```

The numbers after `BRANCH` represent relative jump addresses. `LIT` is the primitive word for pushing a "literal" number onto the data stack.

Compilation state and interpretation state

The word `:` (colon) parses a name as a parameter, creates a dictionary entry (a *colon definition*) and enters compilation state. The interpreter continues to read space-delimited words from the user input device. If a word is found, the interpreter executes the *compilation semantics* associated with the word, instead of the *interpretation semantics*. The default compilation semantics of a word are to append its interpretation semantics to the current definition.

The word `;` (semi-colon) finishes the current definition and returns to interpretation state. It is an example of a word whose compilation semantics differ from the default. The interpretation semantics of `;` (semi-colon), most control flow words, and several other words are undefined in ANS Forth, meaning that they must only be used inside of definitions and not on the interactive command line.

The interpreter state can be changed manually with the words `[` (left-bracket) and `]` (right-bracket) which enter interpretation state or compilation state, respectively. These words can be used with the word `LITERAL` to calculate a value during a compilation and to insert the calculated value into the current colon definition. `LITERAL` has the compilation semantics to take an object from the data stack and to append semantics to the current colon definition to place that object on the data stack.

In ANS Forth, the current state of the interpreter can be read from the flag `STATE` which contains the value true when in compilation state and false otherwise. This allows the implementation of so-called *state-smart words* with behavior that changes according to the current state of the interpreter.

Immediate words

The word `IMMEDIATE` marks the most recent colon definition as an *immediate word*, effectively replacing its compilation semantics with its interpretation semantics. Immediate words are normally executed during compilation, not compiled but this can be overridden by the programmer, in either state. `;` is an example of an immediate word. In ANS Forth, the word `POSTPONE` takes a name as a parameter and appends the compilation semantics of the named word to the current definition even if the word was marked immediate. Forth-83 defined separate words `COMPILE` and `[COMPILE]` to force the compilation of non-immediate and immediate words, respectively.

Unnamed words and execution tokens

In ANS Forth, unnamed words can be defined with the word `:NONAME` which compiles the following words up to the next `;` (semi-colon) and leaves an *execution token* on the data stack. The execution token provides an opaque handle for the compiled semantics, similar to the function pointers of the C programming language.

Execution tokens can be stored in variables. The word `EXECUTE` takes an execution token from the data stack and performs the associated semantics. The word `COMPILE,` (compile-comma) takes an execution token from the data stack and appends the associated semantics to the current definition.

The word `'` (tick) takes the name of a word as a parameter and returns the execution token associated with that word on the data stack. In interpretation state, `' RANDOM-WORD EXECUTE` is equivalent to `RANDOM-WORD`.

Parsing words and comments

The words `:` (colon), `POSTPONE`, `'` (tick) and `:NONAME` are examples of *parsing words* that take their arguments from the user input device instead of the data stack. Another example is the word `(` (paren) which reads and ignores the following words up to and including the next right parenthesis and is used to place comments in a colon definition. Similarly, the word `\` (backslash) is used for comments that continue to the end of the current line. To be parsed correctly, `(` (paren) and `\` (backslash) must be separated by whitespace from the following comment text.

Structure of code

In most Forth systems, the body of a code definition consists of either machine language, or some form of threaded code. The original Forth which follows the informal FIG standard (Forth Interest Group), is a TIL (Threaded Interpretive Language). This is also called indirect-threaded code, but direct-threaded and subroutine threaded Forths have also become popular in modern times. The fastest modern Forths use subroutine threading, insert simple words as macros, and perform peephole optimization or other optimizing strategies to make the code smaller and faster.

Data objects

When a word is a variable or other data object, the CF points to the runtime code associated with the defining word that created it. A defining word has a characteristic "defining behavior" (creating a dictionary entry plus possibly allocating and initializing data space) and also specifies the behavior of an instance of the class of words constructed by this defining word. Examples include:

`VARIABLE`
: Names an uninitialized, one-cell memory location. Instance behavior of a `VARIABLE` returns its address on the stack.

`CONSTANT`
: Names a value (specified as an argument to `CONSTANT`). Instance behavior returns the value.

`CREATE`
: Names a location; space may be allocated at this location, or it can be set to contain a string or other initialized value. Instance behavior returns the address of the beginning of this space.

Forth also provides a facility by which a programmer can define new application-specific defining words, specifying both a custom defining behavior and instance behavior. Some examples include circular buffers, named bits on an I/O port, and automatically-indexed arrays.

Data objects defined by these and similar words are global in scope. The function provided by local variables in other languages is provided by the data stack in Forth (although Forth also has real local variables). Forth programming style uses very few named data objects compared with other languages; typically such data objects are used to contain data which is used by a number of words or tasks (in a multitasked implementation).

Forth does not enforce consistency of data type usage; it is the programmer's responsibility to use appropriate operators to fetch and store values or perform other operations on data.

Programming

Words written in Forth are compiled into an executable form. The classical "indirect threaded" implementations compile lists of addresses of words to be executed in turn; many modern systems generate actual machine code (including calls to some external words and code for others expanded in place). Some systems have optimizing compilers. Generally speaking, a Forth program is saved as the memory image of the compiled program with a single command (e.g., RUN) that is executed when the compiled version is loaded.

During development, the programmer uses the interpreter to execute and test each little piece as it is developed. Most Forth programmers therefore advocate a loose top-down design, and bottom-up development with continuous testing and integration.

The top-down design is usually separation of the program into "vocabularies" that are then used as high-level sets of tools to write the final program. A well-designed Forth program reads like natural language, and implements not just a single solution, but also sets of tools to attack related problems.

Code examples

Hello world

One possible implementation:

```
: HELLO  ( -- )  CR ." Hello, world!" ; HELLO <cr>
Hello, world!
```

The word `CR` (Carriage Return) causes the following output to be displayed on a new line. The parsing word `."` (dot-quote) reads a double-quote delimited string and appends code to the current definition so that the parsed string will be displayed on execution. The space character separating the word `."` from the string `Hello, world!` is not included as part of the string. It is needed so that the parser recognizes `."` as a Forth word.

A standard Forth system is also an interpreter, and the same output can be obtained by typing the following code fragment into the Forth console:

```
CR .( Hello, world!)
```

`.(` (dot-paren) is an immediate word that parses a parenthesis-delimited string and displays it. As with the word `."` the space character separating `.(` from `Hello, world!` is not part of the string.

The word `CR` comes before the text to print. By convention, the Forth interpreter does not start output on a new line. Also by convention, the interpreter waits for input at the end of the previous line, after an `ok` prompt. There is no implied 'flush-buffer' action in Forth's `CR`, as sometimes is in other programming languages.

Mixing compilation state and interpretation state

Here is the definition of a word `EMIT-Q` which when executed emits the single character `Q`:

```
: EMIT-Q   81 ( the ASCII value for the character 'Q' ) EMIT ;
```

This definition was written to use the ASCII value of the `Q` character (81) directly. The text between the parentheses is a comment and is ignored by the compiler. The word `EMIT` takes a value from the data stack and displays the corresponding character.

The following redefinition of `EMIT-Q` uses the words `[` (left-bracket), `]` (right-bracket), `CHAR` and `LITERAL` to temporarily switch to interpreter state, calculate the ASCII value of the `Q` character, return to compilation state and append the calculated value to the current colon definition:

```
: EMIT-Q   [ CHAR Q ]  LITERAL  EMIT ;
```

The parsing word `CHAR` takes a space-delimited word as parameter and places the value of its first character on the data stack. The word `[CHAR]` is an immediate version of `CHAR`. Using `[CHAR]`, the example definition for `EMIT-Q` could be rewritten like this:

```
: EMIT-Q   [CHAR] Q  EMIT ; \ Emit the single character 'Q'
```

This definition used `\` (backslash) for the describing comment.

Both `CHAR` and `[CHAR]` are predefined in ANS Forth. Using `IMMEDIATE` and `POSTPONE`, `[CHAR]` could have been defined like this:

```
: [CHAR]   CHAR  POSTPONE LITERAL ; IMMEDIATE
```

A complete RC4 cipher program

In 1987 Ron Rivest developed the RC4 cipher-system for RSA Data Security, Inc. The code is extremely simple and can be written by most programmers from the description.

We have an array of 256 bytes, all different. Every time the array is used it changes by swapping two bytes. The swaps are controlled by counters *i* and *j*, each initially 0. To get a new *i*, add 1. To get a new *j*, add the array byte at the new *i*. Exchange the array bytes at *i* and *j*. The code is the array byte at the sum of the array bytes at *i* and *j*. This is XORed with a byte of the plaintext to encrypt, or the ciphertext to decrypt. The array is initialized by first setting it to 0 through 255. Then step through it using *i* and *j*, getting the new *j* by adding to it the array byte at *i* and a key byte, and swapping the array bytes at *i* and *j*. Finally, *i* and *j* are set to 0. All additions are modulo 256.

The following Standard Forth version uses Core words only.

```
0 VALUE ii
0 VALUE jj
CREATE S[] 256 CHARS ALLOT
 : ARCFOUR  ( c -- x )
        ii 1+ DUP TO ii 255 AND                ( -- i )
        S[] +  DUP C@                  ( -- 'S[i] S[i] )
        DUP jj +  255 AND DUP TO jj    ( -- 'S[i] S[i] j )
        S[] +  DUP C@ >R               ( -- 'S[i] S[i] 'S[j] )
        OVER SWAP C!                   ( -- 'S[i] S[i] )
        R@ ROT C!                      ( -- S[i] )
        R> +                           ( -- S[i]+S[j] )
        255 AND S[] + C@               ( -- c x )
        XOR ;
: ARCFOUR-INIT  ( key len -- )
        256 MIN LOCALS| len key |
        256 0 DO  I  S[] I + C!  LOOP
        0 TO jj
        256 0  DO                      ( key len -- )
                  key I len MOD + C@
                  S[] I  + C@  + jj + 255 AND TO jj
                  S[] I  + DUP C@  SWAP ( c1 addr1 )
                  S[] jj + DUP C@  ( c1 addr1 addr2 c2 ) ROT C! C!
             LOOP
        0 TO ii  0 TO jj ;
```

This is one of many tests to validate the code.

```
CREATE KEY: 64 CHARS ALLOT
: !KEY ( c1 c2 ... cn n—store the specified key of length n )
        DUP 63 U> ABORT" key too long (<64)"
        DUP KEY: C! KEY: + KEY: 1+ SWAP ?DO  I C!  -1 +LOOP ;

        HEX  61 8A 63 D2 FB  5 !KEY
        KEY: COUNT ARCFOUR-INIT
        CR
           DC ARCFOUR 2 .R SPACE
           EE ARCFOUR 2 .R SPACE
           4C ARCFOUR 2 .R SPACE
           F9 ARCFOUR 2 .R SPACE
           2C ARCFOUR 2 .R
        CR .( Should be: F1 38 29 C9 DE )
```

Implementations

Because the Forth virtual machine is simple to implement and has no standard reference implementation, there are numerous implementations of the language. In addition to supporting the standard varieties of desktop computer systems (POSIX, Microsoft Windows, Mac OS X), many of these Forth systems also target a variety of embedded systems. Listed here are the some of the more prominent systems which conform to the 1994 ANS Forth standard.

Chapter 5

Haskell (Programming Language) and Java (Programming Language)

Haskell (programming language)

Haskell

Paradigm	functional, lazy/non-strict, modular
Appeared in	1990
Designed by	Simon Peyton Jones, Paul Hudak, Philip Wadler, et al.
Stable release	Haskell 2010 (November 24, 2009; 12 months ago)
Preview release	Haskell 2011

Typing discipline	static, strong, inferred
Major implementations	GHC, Hugs, NHC, JHC, Yhc
Dialects	Helium, Gofer
Influenced by	Alfl, APL, FP, Hope, Hope+, Id, ISWIM, KRC, Lisp, Miranda, ML, Standard ML, Lazy ML, Orwell, Ponder, SASL, SISAL, Scheme
Influenced	Agda, Bluespec, Clojure, C#, CAL, Cat, Cayenne, Clean, Curry, Epigram, Escher, F#, Factor, Isabelle, Java Generics, Kaya, LINQ, Mercury, Omega, Perl 6, Python, Qi, Scala, Timber, Visual Basic 9.0
OS	Cross-platform
Usual file extensions	`.hs`, `.lhs`

Haskell is a standardized, general-purpose purely functional programming language, with non-strict semantics and strong static typing. It is named after logician Haskell Curry. In Haskell, "a function is a first-class citizen" of the programming language. As a functional programming language, the primary control construct is the function; the language is rooted in the observations of Haskell Curry and his intellectual descendants, that "a proof is a program; the formula it proves is a type for the program".

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.

Haskell 1.0

The first version of Haskell ("Haskell 1.0") was defined in 1990. The committee's efforts resulted in a series of language definitions.

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.

In February 1999, the Haskell 98 language standard was originally published as "The Haskell 98 Report". In January 2003, a revised version was published as "Haskell 98 Language and Libraries: The Revised Report". 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′** ("Haskell Prime"), was begun. 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.

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 `fak (n+1) = (n+1) * fak 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.

Features

Haskell features lazy evaluation, pattern matching, list comprehensions, typeclasses, 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, but have since found many more uses.

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, and multiple implementations exist.

There is an active community around the language, and more than 2600 third-party open-source libraries and tools are available in the online package repository Hackage.

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, and for having a rich type system incorporating recent innovations such as generalized algebraic data types and Type Families.

Code examples

The following is a Hello world program written in Haskell (note that except for the last line all lines can be omitted):

```
module Main where

main :: IO ()
main = putStrLn "Hello, World!"
```

Here is the factorial function in Haskell, defined in five different ways::

```
-- type
factorial :: Integer -> Integer

-- using recursion
factorial 0 = 1
factorial n = n * factorial (n - 1)

-- using lists
factorial n = product [1..n]

-- using recursion but written without pattern matching
factorial n = if n > 0 then n * factorial (n-1) else 1

-- using fold
factorial n = foldl (*) 1 [1..n]

-- using only prefix notation and n+k-patterns (no longer allowed in
Haskell 2010)
factorial 0 = 1
factorial (n+1) = (*) (n+1) (factorial n)
```

An efficient implementation of the Fibonacci numbers, as an infinite list, is this:

```
-- Point-free style
fib :: Integer -> Integer
fib = (fibs !!)
       where fibs = 0 : scanl (+) 1 fibs

-- Explicit
fib :: Integer -> Integer
fib n = fibs !! n
        where fibs = 0 : scanl (+) 1 fibs
```

Implementations

The following all comply fully, or very nearly, with the Haskell 98 standard, and are distributed under open source licenses. There are currently no proprietary Haskell implementations.

- 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.
- **Gofer** was an educational dialect of Haskell, with a feature called "constructor classes", developed by Mark Jones. It was supplanted by Hugs (see below).
- **HBC** is another native-code Haskell compiler. It has not been actively developed for some time but is still usable.
- **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.
- The **Utrecht Haskell Compiler** (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** 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** 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, the

Haskell tracer. It also featured a JavaScript backend allowing users to run Haskell programs in a web browser.

Applications

Haskell is increasingly being used in commercial situations. 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. Xmonad is a window manager for the X Window System, written entirely in Haskell.

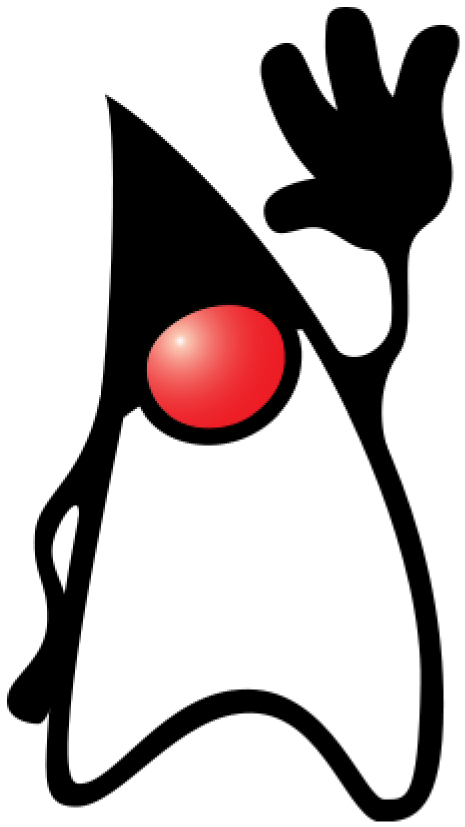
Bluespec SystemVerilog is a language for semiconductor design that is an extension of Haskell. Additionally, Bluespec, Inc.'s tools are implemented in Haskell. Cryptol, a language and toolchain for developing and verifying cryptographic algorithms, is implemented in Haskell. Notably, the first formally verified microkernel, seL4 was verified using Haskell.

Java (programming language)

Java

Paradigm	Object-oriented, structured, imperative
Appeared in	1995
Designed by	Sun Microsystems (now owned by Oracle Corporation)
Developer	James Gosling & Sun Microsystems
Stable release	Java Standard Edition 6 (1.6.0_23) (December 8, 2010; 1 day ago)

Typing discipline	Static, strong, safe, nominative, manifest
Major implementations	OpenJDK, HotSpot, many others
Dialects	Generic Java, Pizza
Influenced by	Ada 83, C++, C#, Delphi Object Pascal, Eiffel, Generic Java, Mesa, Modula-3, Objective-C, UCSD Pascal, Smalltalk
Influenced	Ada 2005, BeanShell, C#, Clojure, D, ECMAScript, Groovy, J#, JavaScript, PHP, Python, Scala
OS	Cross-platform (multi-platform)
License	GNU General Public License / Java Community Process
Usual file extensions	.java, .class, .jar



Duke, the Java mascot

Java is a programming language originally developed by James Gosling at Sun Microsystems (which is now a subsidiary of Oracle Corporation) and released in 1995 as a core component of Sun Microsystems' Java platform. The language derives much of its syntax from C and C++ but has a simpler object model and fewer low-level facilities. Java applications are typically compiled to bytecode (class file) that can run on any Java Virtual Machine (JVM) regardless of computer architecture. Java is a general-purpose, concurrent, class-based, object-oriented language that is specifically designed to have as few implementation dependencies as possible. It is intended to let application developers "write once, run anywhere". Java is currently one of the most popular programming languages in use, and is widely used from application software to web applications.

The original and reference implementation Java compilers, virtual machines, and class libraries were developed by Sun from 1995. As of May 2007, in compliance with the specifications of the Java Community Process, Sun relicensed most of its Java technologies under the GNU General Public License. Others have also developed alternative implementations of these Sun technologies, such as the GNU Compiler for Java, GNU Classpath, and Dalvik.

History

James Gosling, Mike Sheridan, and Patrick Naughton initiated the Java language project in June 1991. Java was originally designed for interactive television, but it was too advanced. The language was initially called *Oak* after an oak tree that stood outside Gosling's office; it went by the name *Green* later, and was later renamed *Java*, from a list of random words. Gosling aimed to implement a virtual machine and a language that had a familiar C/C++ style of notation.

Sun Microsystems released the first public implementation as Java 1.0 in 1995. It promised "Write Once, Run Anywhere" (WORA), providing no-cost run-times on popular platforms. Fairly secure and featuring configurable security, it allowed network- and file-access restrictions. Major web browsers soon incorporated the ability to run Java *applets* within web pages, and Java quickly became popular. With the advent of *Java 2* (released initially as J2SE 1.2 in December 1998–1999), new versions had multiple configurations built for different types of platforms. For example, *J2EE* targeted enterprise applications and the greatly stripped-down version *J2ME* for mobile applications (Mobile Java). *J2SE* designated the Standard Edition. In 2006, for marketing purposes, Sun renamed new *J2* versions as *Java EE*, *Java ME*, and *Java SE*, respectively.

In 1997, Sun Microsystems approached the ISO/IEC JTC1 standards body and later the Ecma International to formalize Java, but it soon withdrew from the process. Java remains a *de facto* standard, controlled through the Java Community Process. At one time, Sun made most of its Java implementations available without charge, despite their proprietary software status. Sun generated revenue from Java through the selling of licenses for specialized products such as the Java Enterprise System. Sun distinguishes between its Software Development Kit (SDK) and Runtime Environment (JRE) (a subset of the SDK); the primary distinction involves the JRE's lack of the compiler, utility programs, and header files.

On November 13, 2006, Sun released much of Java as open source software under the terms of the GNU General Public License (GPL). On May 8, 2007, Sun finished the process, making all of Java's core code available under free software/open-source distribution terms, aside from a small portion of code to which Sun did not hold the copyright.

Sun's vice-president Rich Green has said that Sun's ideal role with regards to Java is as an "evangelist."

Following Oracle Corporation's acquisition of Sun Microsystems in 2009–2010, Oracle has described itself as the "steward of Java technology with a relentless commitment to fostering a community of participation and transparency".

Principles

There were five primary goals in the creation of the Java language:

1. It should be "simple, object oriented, and familiar".
2. It should be "robust and secure".
3. It should be "architecture neutral and portable".
4. It should execute with "high performance".
5. It should be "interpreted, threaded, and dynamic".

Practices

Java Platform

One characteristic of Java is portability, which means that computer programs written in the Java language must run similarly on any supported hardware/operating-system platform. This is achieved by compiling the Java language code to an intermediate representation called Java bytecode, instead of directly to platform-specific machine code. Java bytecode instructions are analogous to machine code, but are intended to be interpreted by a virtual machine (VM) written specifically for the host hardware. End-users commonly use a Java Runtime Environment (JRE) installed on their own machine for standalone Java applications, or in a Web browser for Java applets.

Standardized libraries provide a generic way to access host-specific features such as graphics, threading, and networking.

A major benefit of using bytecode is porting. However, the overhead of interpretation means that interpreted programs almost always run more slowly than programs compiled to native executables would. Just-in-Time compilers were introduced from an early stage that compile bytecodes to machine code during runtime. Over the years, this JVM built-in feature has been optimized to a point where the JVM's performance competes with natively compiled C code.

Implementations

Sun Microsystems officially licenses the Java Standard Edition platform for Linux, Mac OS X, and Solaris. Although in the past Sun has licensed Java to Microsoft, the license has expired and has not been renewed. Through a network of third-party vendors and licensees, alternative Java environments are available for these and other platforms.

Sun's trademark license for usage of the Java brand insists that all implementations be "compatible". This resulted in a legal dispute with Microsoft after Sun claimed that the Microsoft implementation did not support RMI or JNI and had added platform-specific features of their own. Sun sued in 1997, and in 2001 won a settlement of US$20 million, as well as a court order enforcing the terms of the license from Sun. As a result, Microsoft no longer ships Java with Windows, and in recent versions of Windows, Internet Explorer cannot support Java applets without a third-party plugin. Sun, and others, have made available free Java run-time systems for those and other versions of Windows.

Platform-independent Java is essential to the Java EE strategy, and an even more rigorous validation is required to certify an implementation. This environment enables portable server-side applications, such as Web services, Java Servlets, and Enterprise JavaBeans, as well as with embedded systems based on OSGi, using Embedded Java environments. Through the new GlassFish project, Sun is working to create a fully functional, unified open source implementation of the Java EE technologies.

Sun also distributes a superset of the JRE called the Java Development Kit (commonly known as the JDK), which includes development tools such as the Java compiler, Javadoc, Jar, and debugger.

Performance

Programs written in Java have a reputation for being slower and requiring more memory than those written in C. However, Java programs' execution speed improved significantly with the introduction of Just-in-time compilation in 1997/1998 for Java 1.1, the addition of language features supporting better code analysis (such as inner classes, StringBuffer class, optional assertions, etc.), and optimizations in the Java Virtual Machine itself, such as HotSpot becoming the default for Sun's JVM in 2000.

To boost even further the speed performances that can be achieved using the Java language, Systronix made JStik, a microcontroller based on the aJile Systems line of embedded Java processors. In addition, the widely used ARM family of CPUs has hardware support for executing Java bytecode through its Jazelle option.

Automatic memory management

Java uses an automatic garbage collector to manage memory in the object lifecycle. The programmer determines when objects are created, and the Java runtime is responsible for recovering the memory once objects are no longer in use. Once no references to an object remain, the unreachable memory becomes eligible to be freed automatically by the garbage collector. Something similar to a memory leak may still occur if a programmer's code holds a reference to an object that is no longer needed, typically when objects that are no longer needed are stored in containers that are still in use. If methods for a nonexistent object are called, a "null pointer exception" is thrown.

One of the ideas behind Java's automatic memory management model is that programmers can be spared the burden of having to perform manual memory management. In some languages, memory for the creation of objects is implicitly allocated on the stack, or explicitly allocated and deallocated from the heap. In the latter case the responsibility of managing memory resides with the programmer. If the program does not deallocate an object, a memory leak occurs. If the program attempts to access or deallocate memory that has already been deallocated, the result is undefined and difficult to predict, and the program is likely to become unstable and/or crash. This can be partially remedied by the use of smart pointers, but these add overhead and complexity. Note that garbage collection does not prevent "logical" memory leaks, i.e. those where the memory is still referenced but never used.

Garbage collection may happen at any time. Ideally, it will occur when a program is idle. It is guaranteed to be triggered if there is insufficient free memory on the heap to allocate a new object; this can cause a program to stall momentarily. Explicit memory management is not possible in Java.

Java does not support C/C++ style pointer arithmetic, where object addresses and unsigned integers (usually long integers) can be used interchangeably. This allows the garbage collector to relocate referenced objects and ensures type safety and security.

As in C++ and some other object-oriented languages, variables of Java's primitive data types are not objects. Values of primitive types are either stored directly in fields (for objects) or on the stack (for methods) rather than on the heap, as commonly true for objects. This was a conscious decision by Java's designers for performance reasons. Because of this, Java was not considered to be a pure object-oriented programming language. However, as of Java 5.0, autoboxing enables programmers to proceed as if primitive types were instances of their wrapper class.

Syntax

The syntax of Java is largely derived from C++. Unlike C++, which combines the syntax for structured, generic, and object-oriented programming, Java was built almost exclusively as an object-oriented language. All code is written inside a class, and everything is an object, with the exception of the intrinsic data types (ordinal and real numbers, boolean values, and characters), which are not classes for performance reasons.

Java suppresses several features (such as operator overloading and multiple inheritance) for *classes* in order to simplify the language and to prevent possible errors and anti-pattern design.

Java uses similar commenting methods to C++. There are three different styles of comment: a single line style marked with two slashes (//), a multiple line style opened with a slash asterisk (/*) and closed with an asterisk slash (*/), and the Javadoc commenting style opened with a slash and two asterisks (/**) and closed with an asterisk slash (*/). The Javadoc style of commenting allows the user to run the Javadoc executable to compile documentation for the program.

Example:

```
// This is an example of a single line comment using two slashes
```

```
/* This is an example of a multiple line comment using the slash and
asterisk.
   This type of comment can be used to hold a lot of information or
deactivate
   code but it is very important to remember to close the comment. */

/**
 * This is an example of a Javadoc comment; Javadoc can compile
documentation
 *  from this text.
 */
```

Examples

Hello world

The traditional Hello world program can be written in Java as:

```
// Outputs "Hello, world!" and then exits
public class HelloWorld {
   public static void main(String[] args) {
       System.out.println("Hello, world!");
   }
}
```

Source files must be named after the public class they contain, appending the suffix `.java`, for example, `HelloWorld.java`. It must first be compiled into bytecode, using a Java compiler, producing a file named `HelloWorld.class`. Only then can it be executed, or 'launched'. The java source file may only contain one public class but can contain multiple classes with less than public access and any number of public inner classes.

A **`class`** that is not declared **`public`** may be stored in any .java file. The compiler will generate a class file for each class defined in the source file. The name of the class file is the name of the class, with *.class* appended. For class file generation, anonymous classes are treated as if their name were the concatenation of the name of their enclosing class, a *$*, and an integer.

The keyword **`public`** denotes that a method can be called from code in other classes, or that a class may be used by classes outside the class hierarchy. The class hierarchy is related to the name of the directory in which the .java file is located.

The keyword **`static`** in front of a method indicates a static method, which is associated only with the class and not with any specific instance of that class. Only static methods can be invoked without a reference to an object. Static methods cannot access any method variables that are not static.

The keyword **`void`** indicates that the main method does not return any value to the caller. If a Java program is to exit with an error code, it must call System.exit() explicitly.

The method name "`main`" is not a keyword in the Java language. It is simply the name of the method the Java launcher calls to pass control to the program. Java classes that run in managed environments such as applets and Enterprise JavaBean do not use or need a `main()` method. A java program may contain multiple classes that have `main` methods, which means that the VM needs to be explicitly told which class to launch from.

The main method must accept an array of **`String`** objects. By convention, it is referenced as **`args`** although any other legal identifier name can be used. Since Java 5, the main method can also use variable arguments, in the form of `public static void main(String... args)`, allowing the main method to be invoked with an arbitrary number of `String` arguments. The effect of this alternate declaration is semantically identical (the `args` parameter is still an array of `String` objects), but allows an alternative syntax for creating and passing the array.

The Java launcher launches Java by loading a given class (specified on the command line or as an attribute in a JAR) and starting its `public static void main(String[])` method. Stand-alone programs must declare this method explicitly. The `String[] args` parameter is an array of `String` objects containing any arguments passed to the class. The parameters to `main` are often passed by means of a command line.

Printing is part of a Java standard library: The **`System`** class defines a public static field called **`out`**. The `out` object is an instance of the `PrintStream` class and provides many methods for printing data to standard out, including **`println(String)`** which also appends a new line to the passed string.

The string "Hello, world!" is automatically converted to a String object by the compiler.

A more comprehensive example

```
// OddEven.java
import javax.swing.JOptionPane;

public class OddEven {
    // "input" is the number that the user gives to the computer
    private int input; // a whole number("int" means integer)

    /*
     * This is the constructor method. It gets called when an object of
the OddEven type
     * is being created.
     */
    public OddEven() {
    /*
     * Code not shown for simplicity.  In most Java programs
constructors can initialize objects
     * with default values, or create other objects that this object
might use to perform its
     * functions.  In some Java programs, the constructor may simply be
an empty function if nothing
     * needs to be initialized prior to the functioning of the object.
In this program's case, an
```

```
     * empty constructor would suffice, even if it is empty. A 
constructor must exist, however if the
     * user doesn't put one in then the compiler will create an empty 
one.
     */
    }

    // This is the main method. It gets called when this class is run 
through a Java interpreter.
    public static void main(String[] args) {
        /*
         * This line of code creates a new instance of this class 
called "number" (also known as an
         * Object) and initializes it by calling the constructor.  The 
next line of code calls
         * the "showDialog()" method, which brings up a prompt to ask 
you for a number
         */
        OddEven number = new OddEven();
        number.showDialog();
    }

    public void showDialog() {
        /*
         * "try" makes sure nothing goes wrong. If something does,
         * the interpreter skips to "catch" to see what it should do.
         */
        try {
            /*
             * The code below brings up a JOptionPane, which is a 
dialog box
             * The String returned by the "showInputDialog()" method is 
converted into
             * an integer, making the program treat it as a number 
instead of a word.
             * After that, this method calls a second method, 
calculate() that will
             * display either "Even" or "Odd."
             */
            input = 
Integer.parseInt(JOptionPane.showInputDialog("Please Enter A Number"));
            calculate();
        } catch (NumberFormatException e) {
            /*
             * Getting in the catch block means that there was a 
problem with the format of
             * the number. Probably some letters were typed in instead 
of a number.
             */
            System.err.println("ERROR: Invalid input. Please type in a 
numerical value.");
        }
    }

    /*
     * When this gets called, it sends a message to the interpreter.
     * The interpreter usually shows it on the command prompt (For
Windows users)
     * or the terminal (For Linux users).(Assuming it's open)
     */
    private void calculate() {
        if (input % 2 == 0) {
            System.out.println("Even");
        } else {
            System.out.println("Odd");
        }
    }
}
```

- The **import** statement imports the **`JOptionPane`** class from the **`javax.swing`** package.
- The **`OddEven`** class declares a single **`private`** field of type **`int`** named **`input`**. Every instance of the `OddEven` class has its own copy of the `input` field. The private declaration means that no other class can access (read or write) the `input` field.
- **`OddEven()`** is a **`public`** constructor. Constructors have the same name as the enclosing class they are declared in, and unlike a method, have no return type. A constructor is used to initialize an object that is a newly created instance of the class.
- The **`calculate()`** method is declared without the `static` keyword. This means that the method is invoked using a specific instance of the `OddEven` class. (The reference used to invoke the method is passed as an undeclared parameter of type `OddEven` named **`this`**.) The method tests the expression `input % 2 == 0` using the **`if`** keyword to see if the remainder of dividing the `input` field belonging to the instance of the class by two is zero. If this expression is true, then it prints **Even**; if this expression is false it prints **Odd**. (The `input` field can be equivalently accessed as `this.input`, which explicitly uses the undeclared `this` parameter.)
- **`OddEven number = new OddEven();`** declares a local object reference variable in the `main` method named `number`. This variable can hold a reference to an object of type `OddEven`. The declaration initializes `number` by first creating an instance of the `OddEven` class, using the **`new`** keyword and the `OddEven()` constructor, and then assigning this instance to the variable.
- The statement **`number.showDialog();`** calls the calculate method. The instance of `OddEven` object referenced by the `number` local variable is used to invoke the method and passed as the undeclared `this` parameter to the `calculate` method.
- **`input = Integer.parseInt(JOptionPane.showInputDialog("Please Enter A Number"));`** is a statement that converts the type of **String** to the primitive data type **int** by using a utility function in the primitive wrapper class **Integer**.

Special classes

Applet

Java applets are programs that are embedded in other applications, typically in a Web page displayed in a Web browser.

```
// Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {

    @Override
    public void paint(Graphics g) {
        g.drawString("Hello, world!", 65, 95);
    }

}
```

The **`import`** statements direct the Java compiler to include the **`javax.swing.JApplet`** and **`java.awt.Graphics`** classes in the compilation. The import statement allows these classes to be referenced in the source code using the *simple class name* (i.e. `JApplet`) instead of the *fully qualified class name* (i.e. `javax.swing.JApplet`).

The `Hello` class **`extends`** (subclasses) the **`JApplet`** (Java Applet) class; the `JApplet` class provides the framework for the host application to display and control the lifecycle of the applet. The `JApplet` class is a JComponent (Java Graphical Component) which provides the applet with the capability to display a graphical user interface (GUI) and respond to user events.

The `Hello` class overrides the **`paintComponent(Graphics)`** method inherited from the `Container` superclass to provide the code to display the applet. The `paintComponent()` method is passed a **`Graphics`** object that contains the graphic context used to display the applet. The `paintComponent()` method calls the graphic context **`drawString(String, int, int)`** method to display the **"Hello, world!"** string at a pixel offset of (**`65, 95`**) from the upper-left corner in the applet's display.

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
"http://www.w3.org/TR/html4/strict.dtd">
<!-- Hello.html -->
<html>
  <head>
    <title>Hello World Applet</title>
  </head>
  <body>
    <applet code="Hello" width="200" height="200">
    </applet>
  </body>
</html>
```

An applet is placed in an HTML document using the **`<applet>`** HTML element. The `applet` tag has three attributes set: **`code="Hello"`** specifies the name of the `JApplet` class and **`width="200" height="200"`** sets the pixel width and height of the applet. Applets may also be embedded in HTML using either the `object` or `embed` element, although support for these elements by Web browsers is inconsistent. However, the `applet` tag is deprecated, so the `object` tag is preferred where supported.

The host application, typically a Web browser, instantiates the **`Hello`** applet and creates an `AppletContext` for the applet. Once the applet has initialized itself, it is added to the AWT display hierarchy. The `paintComponent()` method is called by the AWT event dispatching thread whenever the display needs the applet to draw itself.

Servlet

Java Servlet technology provides Web developers with a simple, consistent mechanism for extending the functionality of a Web server and for accessing existing business systems. Servlets are server-side Java EE components that generate responses (typically HTML pages) to requests (typically HTTP requests) from clients. A servlet can almost be thought of as an applet that runs on the server side—without a face.

```
// Hello.java
import java.io.*;
import javax.servlet.*;

public class Hello extends GenericServlet {
    public void service(ServletRequest request, ServletResponse
response)
            throws ServletException, IOException {
        response.setContentType("text/html");
        final PrintWriter pw = response.getWriter();
        pw.println("Hello, world!");
        pw.close();
    }
}
```

The **`import`** statements direct the Java compiler to include all of the public classes and interfaces from the **`java.io`** and **`javax.servlet`** packages in the compilation.

The **`Hello`** class **`extends`** the **`GenericServlet`** class; the `GenericServlet` class provides the interface for the server to forward requests to the servlet and control the servlet's lifecycle.

The `Hello` class overrides the **`service(ServletRequest, ServletResponse)`** method defined by the `Servlet` interface to provide the code for the service request handler. The `service()` method is passed a **`ServletRequest`** object that contains the request from the client and a **`ServletResponse`** object used to create the response returned to the client. The `service()` method declares that it **`throws`** the exceptions `ServletException` and `IOException` if a problem prevents it from responding to the request.

The **`setContentType(String)`** method in the response object is called to set the MIME content type of the returned data to **"text/html"**. The **`getWriter()`** method in the response returns a **`PrintWriter`** object that is used to write the data that is sent to the client. The **`println(String)`** method is called to write the **"Hello, world!"** string to the response and then the **`close()`** method is called to close the print writer, which causes the data that has been written to the stream to be returned to the client.

JavaServer Pages

JavaServer Pages (JSP) are server-side Java EE components that generate responses, typically HTML pages, to HTTP requests from clients. JSPs embed Java code in an HTML page by using the special delimiters `<%` and `%>`. A JSP is compiled to a Java *servlet*, a Java application in its own right, the first time it is accessed. After that, the generated servlet creates the response.

Swing application

Swing is a graphical user interface library for the Java SE platform. It is possible to specify a different look and feel through the pluggable look and feel system of Swing. Clones of Windows, GTK+ and Motif are supplied by Sun. Apple also provides an Aqua look and feel for Mac OS X. Where prior implementations of these looks and feels may have been considered lacking, Swing in Java SE 6 addresses this problem by using more native GUI widget drawing routines of the underlying platforms.

This example Swing application creates a single window with "Hello, world!" inside:

```
// Hello.java (Java SE 5)
import javax.swing.*;

public class Hello extends JFrame {
    public Hello() {
        super("hello");
        setDefaultCloseOperation(WindowConstants.EXIT_ON_CLOSE);
        add(new JLabel("Hello, world!"));
        pack();
    }

    public static void main(String[] args) {
        new Hello().setVisible(true);
    }
}
```

The first **`import`** includes all of the public classes and interfaces from the **`javax.swing`** package.

The **`Hello`** class **`extends`** the **`JFrame`** class; the `JFrame` class implements a window with a title bar and a close control.

The **`Hello()`** constructor initializes the frame by first calling the superclass constructor, passing the parameter `"hello"`, which is used as the window's title. It then calls the **`setDefaultCloseOperation(int)`** method inherited from `JFrame` to set the default operation when the close control on the title bar is selected to **`WindowConstants.EXIT_ON_CLOSE`** — this causes the `JFrame` to be disposed of when the frame is closed (as opposed to merely hidden), which allows the JVM to exit and the program to terminate. Next, a **`JLabel`** is created for the string **"Hello, world!"** and the **`add(Component)`** method inherited from the `Container` superclass is called to add the label to the frame. The **`pack()`** method inherited from the `Window` superclass is called to size the window and lay out its contents.

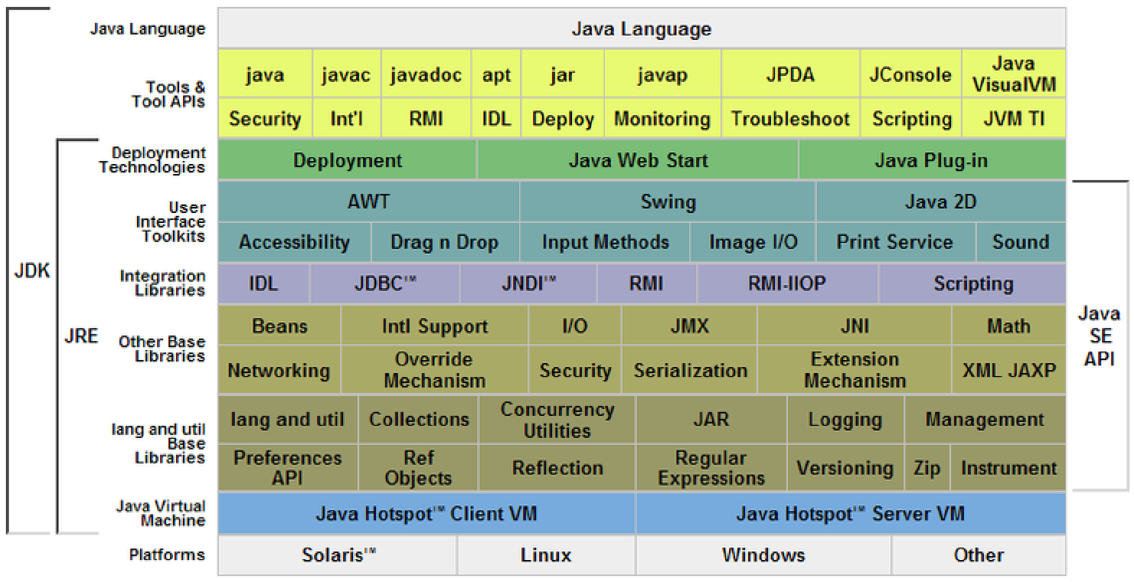
The **`main()`** method is called by the JVM when the program starts. It instantiates a new **`Hello`** frame and causes it to be displayed by calling the **`setVisible(boolean)`** method inherited from the `Component` superclass with the boolean parameter **`true`**. Once the frame is displayed, exiting the `main` method does not cause the program to terminate because the AWT event dispatching thread remains active until all of the Swing top-level windows have been disposed.

Generics

In 2004, generics were added to the Java language, as part of J2SE 5.0. Prior to the introduction of generics, each variable declaration had to be of a specific type. For container classes, for example, this is a problem because there is no easy way to create a container that accepts only specific types of objects. Either the container operates on all subtypes of a class or interface, usually `Object`, or a different container class has to be created for each contained class. Generics allow compile-time type checking without having to create a large number of container classes, each containing almost identical code.

Critical analysis

Class libraries



Java Platform and Class libraries diagram

- Java libraries are the compiled bytecodes of source code developed by the JRE implementor to support application development in Java. Examples of these libraries are:
    - The core libraries, which include:
        - Collection libraries that implement data structures such as lists, dictionaries, trees and sets
        - XML Processing (Parsing, Transforming, Validating) libraries
        - Security
        - Internationalization and localization libraries
    - The integration libraries, which allow the application writer to communicate with external systems. These libraries include:
        - The Java Database Connectivity (JDBC) API for database access
        - Java Naming and Directory Interface (JNDI) for lookup and discovery
        - RMI and CORBA for distributed application development
        - JMX for managing and monitoring applications
    - User interface libraries, which include:
        - The (heavyweight, or native) Abstract Window Toolkit (AWT), which provides GUI components, the means for laying out those components and the means for handling events from those components
        - The (lightweight) Swing libraries, which are built on AWT but provide (non-native) implementations of the AWT widgetry
        - APIs for audio capture, processing, and playback

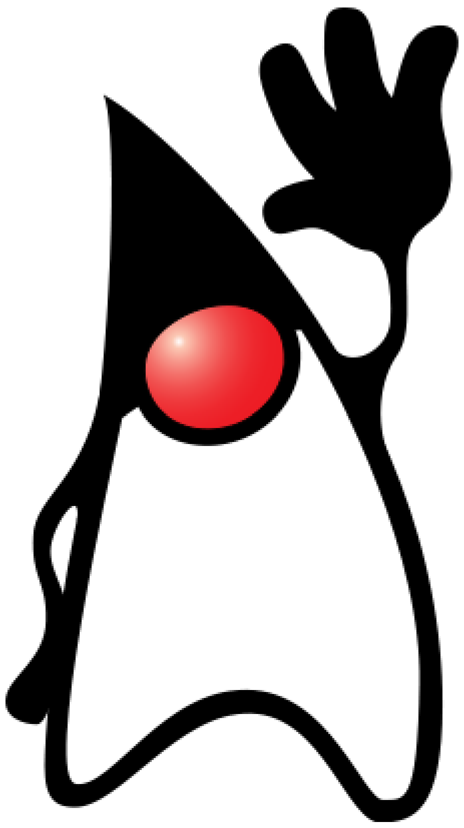
- A platform dependent implementation of Java Virtual Machine (JVM) that is the means by which the byte codes of the Java libraries and third party applications are executed
- Plugins, which enable applets to be run in Web browsers
- Java Web Start, which allows Java applications to be efficiently distributed to end-users across the Internet
- Licensing and documentation.

Documentation

Javadoc is a comprehensive documentation system, created by Sun Microsystems, used by many Java developers. It provides developers with an organized system for documenting their code. Whereas normal comments in Java and C are set off with /* and */, the multi-line comment tags, Javadoc comments have an extra asterisk at the beginning, so that the tags are /** and */.

Editions

Java editions



Java Card

Micro Edition (ME)

Standard Edition (SE)

Enterprise Edition (EE)

PersonalJava (discontinued)

Sun has defined and supports four editions of Java targeting different application environments and segmented many of its APIs so that they belong to one of the platforms. The platforms are:

- Java Card for smartcards.
- Java Platform, Micro Edition (Java ME) — targeting environments with limited resources.
- Java Platform, Standard Edition (Java SE) — targeting workstation environments.
- Java Platform, Enterprise Edition (Java EE) — targeting large distributed enterprise or Internet environments.

The classes in the Java APIs are organized into separate groups called packages. Each package contains a set of related interfaces, classes and exceptions. Refer to the separate platforms for a description of the packages available.

The set of APIs is controlled by Sun Microsystems in cooperation with others through the Java Community Process program. Companies or individuals participating in this process can influence the design and development of the APIs. This process has been a subject of controversy.

Sun also provided an edition called PersonalJava that has been superseded by later, standards-based Java ME configuration-profile pairings.

Chapter 6

Introduction to HDL

In electronics, a **hardware description language** or **HDL** is any language from a class of computer languages, specification languages, or modeling languages for formal description and design of electronic circuits, and most-commonly, digital logic. It can describe the circuit's operation, its design and organization, and tests to verify its operation by means of simulation.

HDLs are standard text-based expressions of the spatial and temporal structure and behaviour of electronic systems. Like concurrent programming languages, HDL syntax and semantics includes explicit notations for expressing concurrency. However, in contrast to most software programming languages, HDLs also include an explicit notion of time, which is a primary attribute of hardware. Languages whose only characteristic is to express circuit connectivity between a hierarchy of blocks are properly classified as netlist languages used on electric computer-aided design (CAD).

HDLs are used to write executable specifications of some piece of hardware. A simulation program, designed to implement the underlying semantics of the language statements, coupled with simulating the progress of time, provides the hardware designer with the ability to model a piece of hardware before it is created physically. It is this executability that gives HDLs the illusion of being programming languages, when they are more-precisely classed as specification languages or modeling languages. Simulators capable of supporting discrete-event (digital) and continuous-time (analog) modeling exist, and HDLs targeted for each are available.

It is certainly possible to represent hardware semantics using traditional programming languages such as C++, although to function such programs must be augmented with extensive and unwieldy class libraries. Primarily, however, software programming languages do not include any capability for explicitly expressing time, and this is why they do not function as a hardware description language. Before the recent introduction of SystemVerilog, C++ integration with a logic simulator was one of the few ways to use OOP in hardware verification. SystemVerilog is the first major HDL to offer object orientation and garbage collection.

Using the proper subset of virtually any (hardware description or software programming) language, a software program called a synthesizer (or synthesis tool) can infer hardware logic operations from the language statements and produce an equivalent netlist of generic hardware primitives to implement the specified behaviour. Synthesizers generally ignore the expression of any timing constructs in the text. Digital logic synthesizers, for example, generally use clock edges as the way to time the circuit, ignoring any timing constructs. The ability to have a synthesizable subset of the language does not itself make a hardware description language.

History

The first hardware description languages were ISP (Instruction Set Processor), developed at Carnegie Mellon University, and KARL, developed at University of Kaiserslautern, both around 1977. ISP was, however, more like a software programming language used to describe relations between the inputs and the outputs of the design. Therefore, it could be used to simulate the design, but not to synthesize it. KARL included design calculus language features supporting VLSI chip floorplanning and structured hardware design, which was also the basis of KARL's interactive graphic sister language ABL, implemented in the early 1980s as the ABLED graphic VLSI design editor, by the telecommunication research center CSELT at Torino, Italy. In the mid 80's, a VLSI design framework was implemented around KARL and ABL by an international consortium funded by the commission of the European Union (chapter in). In 1983 Data-I/O introduced ABEL. It was targeted for describing programmable logical devices and was basically used to design finite state machines.

The first modern HDL, Verilog, was introduced by Gateway Design Automation in 1985. Cadence Design Systems later acquired the rights to Verilog-XL, the HDL-simulator that would become the de-facto standard (of Verilog simulators) for the next decade. In 1987, a request from the U.S. Department of Defense led to the development of VHDL (VHSIC Hardware Description Language, where VHSIC is Very High Speed Integrated Circuit). VHDL was based on the Ada programming language. Initially, Verilog and VHDL were used to document and simulate circuit-designs already captured and described in another form (such as a schematic file.) HDL-simulation enabled engineers to work at a higher level of abstraction than simulation at the schematic-level, and thus increased design capacity from hundreds of transistors to thousands.

The introduction of logic-synthesis for HDLs pushed HDLs from the background into the foreground of digital-design. Synthesis tools compiled HDL-source files (written in a constrained format called RTL) into a manufacturable gate/transistor-level netlist description. Writing synthesizeable RTL files required practice and discipline on the part of the designer; compared to a traditional schematic-layout, synthesized-RTL netlists were almost always larger in area and slower in performance. Circuit design by a skilled engineer, using labor-intensive schematic-capture/hand-layout, would almost always outperform its logically-synthesized equivalent, but synthesis's productivity advantage soon displaced digital schematic-capture to exactly those areas that were problematic for RTL-synthesis: extremely high-speed, low-power, or asynchronous circuitry. In short, logic synthesis propelled HDL technology into a central role for digital design.

Within a few years, both VHDL and Verilog emerged as the dominant HDLs in the electronics industry, while older and less-capable HDLs gradually disappeared from use. But VHDL and Verilog share many of the same limitations: neither HDL is suitable for analog/mixed-signal circuit simulation. Neither possesses language constructs to describe recursively-generated logic structures. Specialized HDLs (such as Confluence) were introduced with the explicit goal of fixing a specific Verilog/VHDL limitation, though none were ever intended to replace VHDL/Verilog.

Over the years, a lot of effort has gone into improving HDLs. The latest iteration of Verilog, formally known as IEEE 1800-2005 SystemVerilog, introduces many new features (classes, random variables, and properties/assertions) to address the growing need for better testbench randomization, design hierarchy, and reuse. A future revision of VHDL is also in development, and is expected to match SystemVerilog's improvements.

Design using HDL

Efficiency gains realized using HDL means a majority of modern digital circuit design revolves around it. Most designs begin as a set of requirements or a high-level architectural diagram. Control and decision structures are often prototyped in flowchart applications, or entered in a state-diagram editor. The process of writing the HDL description is highly dependent on the nature of the circuit and the designer's preference for coding style . The HDL is merely the 'capture language'—often beginning with a high-level algorithmic description such as MATLAB or a C++ mathematical model. Designers often use scripting languages (such as Perl) to automatically generate repetitive circuit structures in the HDL language. Special text editors offer features for automatic indentation, syntax-dependent coloration, and macro-based expansion of entity/architecture/signal declaration.

The HDL code then undergoes a code review, or auditing. In preparation for synthesis, the HDL description is subject to an array of automated checkers. The checkers report deviations from standardized code guidelines, identify potential ambiguous code constructs before they can cause misinterpretation, and check for common logical coding errors, such as dangling ports or shorted outputs. This process aids in resolving errors before the code is synthesized.

In industry parlance, HDL design generally ends at the synthesis stage. Once the synthesis tool has mapped the HDL description into a gate netlist, this netlist is passed off to the back-end stage. Depending on the physical technology (FPGA, ASIC gate array, ASIC standard cell), HDLs may or may not play a significant role in the back-end flow. In general, as the design flow progresses toward a physically realizable form, the design database becomes progressively more laden with technology-specific information, which cannot be stored in a generic HDL description. Finally, an integrated circuit is manufactured or programmed for use.

Simulating and debugging HDL code

Essential to HDL design is the ability to simulate HDL programs. Simulation allows an HDL description of a design (called a model) to pass design verification, an important milestone that validates the design's intended function (specification) against the code implementation in the HDL description. It also permits architectural exploration. The engineer can experiment with design choices by writing multiple variations of a base design, then comparing their behavior in simulation. Thus, simulation is critical for successful HDL design.

To simulate an HDL model, an engineer writes a top-level simulation environment (called a testbench). At minimum, a testbench contains an instantiation of the model (called the device under test or DUT), pin/signal declarations for the model's I/O, and a clock waveform. The testbench code is event driven: the engineer writes HDL statements to implement the (testbench-generated) reset-signal, to model interface transactions (such as a host–bus read/write), and to monitor the DUT's output. An HDL simulator — the program that executes the testbench — maintains the simulator clock, which is the master reference for all events in the testbench simulation. Events occur only at the instants dictated by the testbench HDL (such as a reset-toggle coded into the testbench), or in reaction (by the model) to stimulus and triggering events. Modern HDL simulators have a full-featured graphical user interfaces, complete with a suite of debug tools. These allow the user to stop and restart the simulation at any time, insert simulator breakpoints (independent of the HDL code), and monitor or modify any element in the HDL model hierarchy. Modern simulators can also link the HDL environment to user-compiled libraries, through a defined PLI/VHPI interface. Linking is system-dependent (Win32/Linux/SPARC), as the HDL simulator and user libraries are compiled and linked outside the HDL environment.

Design verification is often the most time-consuming portion of the design process, due to the disconnect between a device's functional specification, the designer's interpretation of the specification, and the imprecision of the HDL language. The majority of the initial test/debug cycle is conducted in the HDL *simulator* environment, as the early stage of the design is subject to frequent and major circuit changes. An HDL description can also be prototyped and tested in hardware — programmable logic devices are often used for this purpose. Hardware prototyping is comparatively more expensive than HDL simulation, but offers a real-world view of the design. Prototyping is the best way to check interfacing against other hardware devices and hardware prototypes. Even those running on slow FPGAs offer much faster simulation times than pure HDL simulation.

Design Verification with HDLs

Historically, design verification was a laborious, repetitive loop of writing and running simulation test cases against the design under test. As chip designs have grown larger and more complex, the task of design verification has grown to the point where it now dominates the schedule of a design team. Looking for ways to improve design productivity, the EDA industry developed the Property Specification Language.

In formal verification terms, a property is a factual statement about the expected or assumed behavior of another object. Ideally, for a given HDL description, a property or properties can be proven true or false using formal mathematical methods. In practical terms, many properties cannot be proven because they occupy an unbounded solution space. However, if provided a set of operating assumptions or constraints, a property checker can prove (or disprove) more properties, over the narrowed solution space.

The assertions do not model circuit activity, but capture and document the "designer's intent" in the HDL code. In a simulation environment, the simulator evaluates all specified assertions, reporting the location and severity of any violations. In a synthesis environment, the synthesis tool usually operates with the policy of halting synthesis upon any violation. Assertion-based verification is still in its infancy, but is expected to become an integral part of the HDL design toolset.

HDL and programming languages

A HDL is analogous to a software programming language, but with major differences. Many programming languages are inherently procedural (single-threaded), with limited syntactical and semantic support to handle concurrency. HDLs, on the other hand, resemble concurrent programming languages in their ability to model multiple parallel processes (such as flipflops, adders, etc.) that automatically execute independently of one another. Any change to the process's input automatically triggers an update in the simulator's process stack. Both programming languages and HDLs are processed by a compiler (usually called a synthesizer in the HDL case), but with different goals. For HDLs, 'compiler' refers to synthesis, a process of transforming the HDL code listing into a physically realizable gate netlist. The netlist output can take any of many forms: a "simulation" netlist with gate-delay information, a "handoff" netlist for post-synthesis place and route, or a generic industry-standard EDIF format (for subsequent conversion to a JEDEC-format file).

On the other hand, a software compiler converts the source-code listing into a microprocessor-specific object-code, for execution on the target microprocessor. As HDLs and programming languages borrow concepts and features from each other, the boundary between them is becoming less distinct. However, pure HDLs are unsuitable for general purpose software application development, just as general-purpose programming languages are undesirable for modeling hardware. Yet as electronic systems grow increasingly complex, and reconfigurable systems become increasingly mainstream, there is growing desire in the industry for a single language that can perform some tasks of both hardware design and software programming. SystemC is an example of such—embedded system hardware can be modeled as non-detailed architectural blocks (blackboxes with modeled signal inputs and output drivers). The target application is written in C/C++, and natively compiled for the host-development system (as opposed to targeting the embedded CPU, which requires host-simulation of the embedded CPU). The high level of abstraction of SystemC models is well suited to early architecture exploration, as architectural modifications can be easily evaluated with little concern for signal-level implementation issues. However, the threading model used in SystemC and its reliance on shared memory mean that it does not handle parallel execution or lower level models well.

In an attempt to reduce the complexity of designing in HDLs, which have been compared to the equivalent of assembly languages, there are moves to raise the abstraction level of the design. Companies such as Cadence, Synopsys and Agility Design Solutions are promoting SystemC as a way to combine high level languages with concurrency models to allow faster design cycles for FPGAs than is possible using traditional HDLs. Approaches based on standard C or C++ (with libraries or other extensions allowing parallel programming) are found in the Catapult C tools from Mentor Graphics, the Impulse C tools from Impulse Accelerated Technologies, and the free and open-source ROCCC 2.0 tools from Jacquard Computing Inc. Annapolis Micro Systems, Inc.'s CoreFire Design Suite and National Instruments LabVIEW FPGA provide a graphical dataflow approach to high-level design entry. Languages such as SystemVerilog, SystemVHDL, and Handel-C seek to accomplish the same goal, but are aimed at making existing hardware engineers more productive versus making FPGAs more accessible to existing software engineers. There is more information on C to HDL and Flow to HDL in their respective articles.

Languages

Analogue circuit design

Abbreviation	**Name**	**Use**
AHDL	Analog Hardware Descriptive Language (HDL)	an open analog hardware description language
SpectreHDL	SpectreHDL	a proprietary analogue hardware description language
Verilog-AMS	Verilog for Analog and Mixed-Signal	an open standard extending Verilog for analog and mixed analog/digital simulation
HDL-A™	HDL-A	a proprietary analogue hardware description language

Digital circuit design

The two most widely-used and well-supported HDL varieties used in industry are Verilog and VHDL.

Abbreviation	**Name**	**Notice**
ABEL	Advanced Boolean Expression Language	
AHDL	Altera HDL	a proprietary language from Altera)
AHPL	A Hardware Programing language	

Bluespec		high-level HDL originally based on Haskell, now with a SystemVerilog syntax
C-to-Verilog		Converter from C to Verilog
Confluence		a functional HDL; has been discontinued)
CoWareC		a C-based HDL by CoWare. Now discontinued in favor of SystemC
CUPL		a proprietary language from Logical Devices, Inc.)
ELLA		no longer in common use)
Handel-C		a C-like design language
HJJ	Hardware Join Java	based on Join Java)
HML		based on SML
Hydra		based on Haskell
Impulse C	another C-like HDL	
ParC	Parallel C++	C++ extended with HDL style threading and communication for task-parallel programming
JHDL		based on Java)
Lava		based on Haskell
Lola		a simple language used for teaching
M		A HDL from Mentor Graphics
MyHDL		based on Python
PALASM		for Programmable Array Logic (PAL) devices
ROCCC 2.0	Riverside Optimizing Compiler for Configurable Computing	Free and open-source C to HDL tool
RHDL		based on the Ruby programming language)
Ruby (hardware description language)		
SystemC		a standardized class of C++ libraries for high-level behavioral and transaction modeling of digital hardware at a high level of abstraction, i.e. system-level
SystemVerilog		a superset of Verilog, with enhancements to address system-level design and verification
SystemTCL		SDL based on Tcl.
Verilog		most widely-used and well-supported HDL
VHDL	VHSIC HDL	most widely-used and well-supported HDL

Chapter 7

VHDL

	VHDL
Paradigm	concurrent, reactive
Appeared in	1980s
Typing discipline	strong
Influenced by	Ada, Pascal

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;

entity signed_adder is
  port
  (
    aclr : in    std_logic;
    clk  : in    std_logic;
    a    : in    std_logic_vector;
    b    : in    std_logic_vector;
    q    : out   std_logic_vector
  );
end signed_adder;

architecture signed_adder_arch of signed_adder is
  signal q_s : signed(a'high+1 downto 0); -- extra bit wide

begin -- architecture
  assert (a'length >= b'length)
    report "Port A must be the longer vector if different sizes!"
    severity FAILURE;
  q <= std_logic_vector(q_s);

  adding_proc:
  process (aclr, clk)
    begin
      if (aclr = '1') then
        q_s <= (others => '0');
      elsif rising_edge(clk) then
        q_s <= ('0'&signed(a)) + ('0'&signed(b));
      end if; -- clk'd
    end process;

end signed_adder_arch;
```

VHDL source for a signed adder

VHDL (**VHSIC hardware description language**) is a hardware description language used in electronic design automation to describe digital and mixed-signal systems such as field-programmable gate arrays and integrated circuits.

History

VHDL was originally developed at the behest of the U.S Department of Defense in order to document the behavior of the ASICs that supplier companies were including in equipment. That is to say, VHDL was developed as an alternative to huge, complex manuals which were subject to implementation-specific details.

The idea of being able to simulate this documentation was so obviously attractive that logic simulators were developed that could read the VHDL files. The next step was the development of logic synthesis tools that read the VHDL, and output a definition of the physical implementation of the circuit.

Due to the Department of Defense requiring as much of the syntax as possible to be based on Ada, in order to avoid re-inventing concepts that had already been thoroughly tested in the development of Ada, VHDL borrows heavily from the Ada programming language in both concepts and syntax.

The initial version of VHDL, designed to IEEE standard 1076-1987, included a wide range of data types, including numerical (integer and real), logical (bit and boolean), character and time, plus arrays of `bit` called `bit_vector` and of `character` called string.

A problem not solved by this edition, however, was "multi-valued logic", where a signal's drive strength (none, weak or strong) and unknown values are also considered. This required IEEE standard 1164, which defined the 9-value logic types: scalar `std_ulogic` and its vector version `std_ulogic_vector`.

The updated IEEE 1076, in 1993, made the syntax more consistent, allowed more flexibility in naming, extended the `character` type to allow ISO-8859-1 printable characters, added the `xnor` operator, etc.

Minor changes in the standard (2000 and 2002) added the idea of protected types (similar to the concept of class in C++) and removed some restrictions from port mapping rules.

In addition to IEEE standard 1164, several child standards were introduced to extend functionality of the language. IEEE standard 1076.2 added better handling of real and complex data types. IEEE standard 1076.3 introduced signed and unsigned types to facilitate arithmetical operations on vectors. IEEE standard 1076.1 (known as VHDL-AMS) provided analog and mixed-signal circuit design extensions.

Some other standards support wider use of VHDL, notably VITAL (VHDL Initiative Towards ASIC Libraries) and microwave circuit design extensions.

In June 2006, the VHDL Technical Committee of Accellera (delegated by IEEE to work on the next update of the standard) approved so called Draft 3.0 of VHDL-2006. While maintaining full compatibility with older versions, this proposed standard provides numerous extensions that make writing and managing VHDL code easier. Key changes include incorporation of child standards (1164, 1076.2, 1076.3) into the main 1076 standard, an extended set of operators, more flexible syntax of *case* and *generate* statements, incorporation of VHPI (interface to C/C++ languages) and a subset of PSL (Property Specification Language). These changes should improve quality of synthesizable VHDL code, make testbenches more flexible, and allow wider use of VHDL for system-level descriptions.

In February 2008, Accellera approved VHDL 4.0 also informally known as VHDL 2008, which addressed more than 90 issues discovered during the trial period for version 3.0 and includes enhanced generic types. In 2008, Accellera released VHDL 4.0 to the IEEE for balloting for inclusion in IEEE 1076-2008. The VHDL standard IEEE 1076-2008 was published in January 2009.

Design

VHDL is commonly used to write text models that describe a logic circuit. Such a model is processed by a synthesis program, only if it is part of the logic design. A simulation program is used to test the logic design using simulation models to represent the logic circuits that interface to the design. This collection of simulation models is commonly called a *testbench*.

VHDL has constructs to handle the parallelism inherent in hardware designs, but these constructs (*processes*) differ in syntax from the parallel constructs in Ada (*tasks*). Like Ada, VHDL is strongly typed and is not case sensitive. In order to directly represent operations which are common in hardware, there are many features of VHDL which are not found in Ada, such as an extended set of Boolean operators including **nand** and **nor**. VHDL also allows arrays to be indexed in either ascending or descending direction; both conventions are used in hardware, whereas in Ada and most programming languages only ascending indexing is available.

VHDL has file input and output capabilities, and can be used as a general-purpose language for text processing, but files are more commonly used by a simulation testbench for stimulus or verification data. There are some VHDL compilers which build executable binaries. In this case, it might be possible to use VHDL to write a *testbench* to verify the functionality of the design using files on the host computer to define stimuli, to interact with the user, and to compare results with those expected. However, most designers leave this job to the simulator.

It is relatively easy for an inexperienced developer to produce code that simulates successfully but that cannot be synthesized into a real device, or is too large to be practical. One particular pitfall is the accidental production of transparent latches rather than D-type flip-flops as storage elements.

One can design hardware in a VHDL IDE (for FPGA implementation such as Xilinx ISE, Altera Quartus, Synopsys Synplify or Mentor Graphics HDL Designer) to produce the RTL schematic of the desired circuit. After that, the generated schematic can be verified using simulation software which shows the waveforms of inputs and outputs of the circuit after generating the appropriate testbench. To generate an appropriate testbench for a particular circuit or VHDL code, the inputs have to be defined correctly. For example, for clock input, a loop process or an iterative statement is required.

A final point is that when a VHDL model is translated into the "gates and wires" that are mapped onto a programmable logic device such as a CPLD or FPGA, then it is the actual hardware being configured, rather than the VHDL code being "executed" as if on some form of a processor chip.

Advantages

The key advantage of VHDL, when used for systems design, is that it allows the behavior of the required system to be described (modeled) and verified (simulated) before synthesis tools translate the design into real hardware (gates and wires).

Another benefit is that VHDL allows the description of a concurrent system. VHDL is a dataflow language, unlike procedural computing languages such as BASIC, C, and assembly code, which all run sequentially, one instruction at a time.

VHDL project is multipurpose. Being created once, a calculation block can be used in many other projects. However, many formational and functional block parameters can be tuned (capacity parameters, memory size, element base, block composition and interconnection structure).

VHDL project is portable. Being created for one element base, a computing device project can be ported on another element base, for example VLSI with various technologies.

Design examples

In VHDL, a design consists at a minimum of an *entity* which describes the interface and an *architecture* which contains the actual implementation. In addition, most designs import library modules. Some designs also contain multiple architectures and *configurations*.

A simple AND gate in VHDL would look something like this:

```
-- (this is a VHDL comment)

-- import std_logic from the IEEE library
library IEEE;
use IEEE.std_logic_1164.all;
```

```
-- this is the entity
entity ANDGATE is
   port (
         IN1 : in std_logic;
         IN2 : in std_logic;
         OUT1: out std_logic);
end ANDGATE;

architecture RTL of ANDGATE is
begin

  OUT1 <= IN1 and IN2;

end RTL;
```

While the example above may seem very verbose to HDL beginners, many parts are either optional or need to be written only once. Generally simple functions like this are part of a larger behavioral module, instead of having a separate module for something so simple. In addition, use of elements such as the `std_logic` type might at first seem to be an overkill. One could easily use the built-in `bit` type and avoid the library import in the beginning. However, using this 9-valued logic (`U`,`X`,`0`,`1`,`Z`,`W`,`H`,`L`,`-`) instead of simple bits (0,1) offers a very powerful simulation and debugging tool to the designer which currently does not exist in any other HDL.

In the examples that follow, you will see that VHDL code can be written in a very compact form. However, the experienced designers usually avoid these compact forms and use a more verbose coding style for the sake of readability and maintainability. Another advantage to the verbose coding style is the smaller amount of resources used when programming to a Programmable Logic Device such as a CPLD.

Synthesizeable constructs and VHDL templates

VHDL is frequently used for two different goals: simulation of electronic designs and synthesis of such designs. Synthesis is a process where a VHDL is compiled and mapped into an implementation technology such as an FPGA or an ASIC. Many FPGA vendors have free (or inexpensive) tools to synthesize VHDL for use with their chips, where ASIC tools are often very expensive.

Not all constructs in VHDL are suitable for synthesis. For example, most constructs that explicitly deal with timing such as `wait for 10 ns;` are not synthesizable despite being valid for simulation. While different synthesis tools have different capabilities, there exists a common *synthesizable subset* of VHDL that defines what language constructs and idioms map into common hardware for many synthesis tools. IEEE 1076.6 defines a subset of the language that is considered the official synthesis subset. It is generally considered a "best practice" to write very idiomatic code for synthesis as results can be incorrect or suboptimal for non-standard constructs.

Some examples of synthesizable code follow below:

MUX template

The multiplexer, or 'MUX' as it is usually called, is a simple construct very common in hardware design. The example below demonstrates a simple two to one MUX, with inputs `A` and `B`, selector `S` and output `X`. Note that there are many other ways to express the same MUX in VHDL.

```
X <= A when S = '1' else B;
```

Latch template

A transparent latch is basically one bit of memory which is updated when an enable signal is raised. Again, there are many other ways this can be expressed in VHDL.

```
-- latch template 1:
Q <= D when Enable = '1' else Q;

-- latch template 2:
process(D,Enable)
begin
  if Enable = '1' then
    Q <= D;
  end if;
end process;
```

D-type flip-flops

The D-type flip-flop samples an incoming signal at the rising (or falling edge) of a clock. This example has an asynchronous, active-high reset, and samples at the rising clock edge.

```
process(CLK, RESET)
begin
  if RESET = '1' then
    Q <= '0';
  elsif rising_edge(CLK) then
    Q <= D;
  end if;
end process;
```

Example: a counter

The following example is an up-counter with asynchronous reset, parallel load and configurable width. It demonstrates the use of the 'unsigned' type and VHDL *generics*. The generics are very close to arguments or templates in other traditional programming languages like C or C++.

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;    -- for the unsigned type
```

```
entity counter_example is
generic ( WIDTH : integer := 32);
port (
  CLK, RESET, LOAD : in std_logic;
  DATA : in  unsigned(WIDTH-1 downto 0);
  Q    : out unsigned(WIDTH-1 downto 0));
end entity counter_example;

architecture counter_example_a of counter_example is
signal cnt : unsigned(WIDTH-1 downto 0);
begin
  process(RESET, CLK) is
  begin
    if RESET = '1' then
      cnt <= (others => '0');
    elsif rising_edge(CLK) then
      if LOAD = '1' then
        cnt <= DATA;
      else
        cnt <= cnt + 1;
      end if;
    end if;
  end process;

  Q <= cnt;

end architecture counter_example_a;
```

More complex counters may add if/then/else statements within the `rising_edge(CLK) elsif` to add other functions, such as count enables, stopping or rolling over at some count value, generating output signals like terminal count signals, etc. Care must be taken with the ordering and nesting of such controls if used together, in order to produce the desired priorities and minimize the number of logic levels needed.

Simulation-only constructs

A large subset of VHDL cannot be translated into hardware. This subset is known as the non-synthesizable or the simulation-only subset of VHDL and can only be used for prototyping, simulation and debugging. For example, the following code will generate a clock with the frequency of 50 MHz. It can, for example, be used to drive a clock input in a design during simulation. It is, however, a simulation-only construct and cannot be implemented in hardware. In actual hardware, the clock is generated externally; it can be scaled down internally by user logic or dedicated hardware.

```
process
begin
  CLK <= '1'; wait for 10 ns;
  CLK <= '0'; wait for 10 ns;
end process;
```

The simulation-only constructs can be used to build complex waveforms in very short time. Such waveform can be used, for example, as test vectors for a complex design or as a prototype of some synthesizable logic that will be implemented in future.

```
process
begin
  wait until START = '1'; -- wait until START is high

  for i in 1 to 10 loop -- then wait for a few clock periods...
    wait until rising_edge(CLK);
  end loop;

  for i in 1 to 10 loop     -- write numbers 1 to 10 to DATA, 1 every
cycle
    DATA <= to_unsigned(i, 8);
    wait until rising_edge(CLK);
  end loop;

  -- wait until the output changes
  wait on RESULT;

  -- now raise ACK for clock period
  ACK <= '1';
  wait until rising_edge(CLK);
  ACK <= '0';


  -- and so on...
end process;
```

Chapter 8

Verilog

In the semiconductor and electronic design industry, **Verilog** is a hardware description language (HDL) used to model electronic systems. *Verilog HDL*, not to be confused with VHDL, is most commonly used in the design, verification, and implementation of digital logic chips at the register transfer level (RTL) of abstraction. It is also used in the verification of analog and mixed-signal circuits.

Overview

Hardware description languages such as Verilog, differ from software programming languages because they include ways of describing the propagation of time and signal dependencies (sensitivity). There are two assignment operators, a blocking assignment (=), and a non-blocking (<=) assignment. The non-blocking assignment allows designers to describe a state-machine update without needing to declare and use temporary storage variables (in any general programming language we need to define some temporary storage spaces for the operands to be operated on subsequently; those are temporary storage variables). Since these concepts are part of Verilog's language semantics, designers could quickly write descriptions of large circuits, in a relatively compact and concise form. At the time of Verilog's introduction (1984), Verilog represented a tremendous productivity improvement for circuit designers who were already using graphical schematic capture software and specially-written software programs to document and simulate electronic circuits.

The designers of Verilog wanted a language with syntax similar to the C programming language, which was already widely used in engineering software development. Verilog is case-sensitive, has a basic preprocessor (though less sophisticated than that of ANSI C/C++), and equivalent control flow keywords (if/else, for, while, case, etc.), and compatible operator precedence. Syntactic differences include variable declaration (Verilog requires bit-widths on net/reg types), demarcation of procedural blocks (begin/end instead of curly braces {}), and many other minor differences.

A Verilog design consists of a hierarchy of modules. Modules encapsulate *design hierarchy*, and communicate with other modules through a set of declared input, output, and bidirectional ports. Internally, a module can contain any combination of the following: net/variable declarations (wire, reg, integer, etc.), concurrent and sequential statement blocks, and instances of other modules (sub-hierarchies). Sequential statements are placed inside a begin/end block and executed in sequential order within the block. But the blocks themselves are executed concurrently, qualifying Verilog as a dataflow language.

Verilog's concept of 'wire' consists of both signal values (4-state: "1, 0, floating, undefined"), and strengths (strong, weak, etc.) This system allows abstract modeling of shared signal-lines, where multiple sources drive a common net. When a wire has multiple drivers, the wire's (readable) value is resolved by a function of the source drivers and their strengths.

A subset of statements in the Verilog language are synthesizable. Verilog modules that conform to a synthesizable coding-style, known as RTL (register transfer level), can be physically realized by synthesis software. Synthesis-software algorithmically transforms the (abstract) Verilog source into a netlist, a logically-equivalent description consisting only of elementary logic primitives (AND, OR, NOT, flipflops, etc.) that are available in a specific FPGA or VLSI technology. Further manipulations to the netlist ultimately lead to a circuit fabrication blueprint (such as a photo mask set for an ASIC, or a bitstream file for an FPGA).

History

Beginning

Verilog was invented by Phil Moorby and Prabhu Goel during the winter of 1983/1984 at Automated Integrated Design Systems (renamed to Gateway Design Automation in 1985) as a hardware modeling language. Gateway Design Automation was purchased by Cadence Design Systems in 1990. Cadence now has full proprietary rights to Gateway's Verilog and the Verilog-XL simulator logic simulators. Originally, Verilog was intended to describe and allow simulation; only afterwards was support for synthesis added.

Verilog-95

With the increasing success of VHDL at the time, Cadence decided to make the language available for open standardization. Cadence transferred Verilog into the public domain under the Open Verilog International (OVI) (now known as Accellera) organization. Verilog was later submitted to IEEE and became IEEE Standard 1364-1995, commonly referred to as Verilog-95.

In the same time frame Cadence initiated the creation of Verilog-A to put standards support behind its analog simulator Spectre. Verilog-A was never intended to be a standalone language and is a subset of Verilog-AMS which encompassed Verilog-95.

Verilog 2001

Extensions to Verilog-95 were submitted back to IEEE to cover the deficiencies that users had found in the original Verilog standard. These extensions became IEEE Standard 1364-2001 known as Verilog-2001.

Verilog-2001 is a significant upgrade from Verilog-95. First, it adds explicit support for (2's complement) signed nets and variables. Previously, code authors had to perform signed-operations using awkward bit-level manipulations (for example, the carry-out bit of a simple 8-bit addition required an explicit description of the boolean-algebra to determine its correct value). The same function under Verilog-2001 can be more succinctly described by one of the built-in operators: +, -, /, *, >>>. A generate/endgenerate construct (similar to VHDL's generate/endgenerate) allows Verilog-2001 to control instance and statement instantiation through normal decision-operators (case/if/else). Using generate/endgenerate, Verilog-2001 can instantiate an array of instances, with control over the connectivity of the individual instances. File I/O has been improved by several new system-tasks. And finally, a few syntax additions were introduced to improve code-readability (e.g. always @*, named-parameter override, C-style function/task/module header declaration).

Verilog-2001 is the dominant flavor of Verilog supported by the majority of commercial EDA software packages.

Verilog 2005

Not to be confused with SystemVerilog, *Verilog 2005* (IEEE Standard 1364-2005) consists of minor corrections, spec clarifications, and a few new language features (such as the uwire keyword).

A separate part of the Verilog standard, Verilog-AMS, attempts to integrate analog and mixed signal modeling with traditional Verilog.

SystemVerilog

SystemVerilog is a superset of Verilog-2005, with many new features and capabilities to aid design-verification and design-modeling. As of 2009, the SystemVerilog and Verilog language standards were merged into SystemVerilog 2009 (IEEE Standard 1800-2009).

The advent of hardware verification languages such as OpenVera, and Verisity's e language encouraged the development of Superlog by Co-Design Automation Inc. Co-Design Automation Inc was later purchased by Synopsys. The foundations of Superlog and Vera were donated to Accellera, which later became the IEEE standard P1800-2005: SystemVerilog.

Example

A hello world program looks like this:

```
module main;
  initial
    begin
      $display("Hello world!");
      $finish;
    end
endmodule
```

A simple example of two flip-flops follows:

```
module toplevel(clock,reset);
 input clock;
 input reset;

 reg flop1;
 reg flop2;

 always @ (posedge reset or posedge clock)
 if (reset)
   begin
     flop1 <= 0;
     flop2 <= 1;
   end
 else
   begin
     flop1 <= flop2;
     flop2 <= flop1;
   end
endmodule
```

The "<=" operator in verilog is another aspect of its being a hardware description language as opposed to a normal procedural language. This is known as a "non-blocking" assignment. Its action doesn't register until the next clock cycle. This means that the order of the assignments are irrelevant and will produce the same result: flop1 and flop2 will swap values every clock.

The other assignment operator, "=", is referred to as a blocking assignment. When "=" assignment is used, for the purposes of logic, the target variable is updated immediately. In the above example, had the statements used the "=" blocking operator instead of "<=", flop1 and flop2 would not have been swapped. Instead, as in traditional programming, the compiler would understand to simply set flop1 equal to flop2.

An example counter circuit follows:

```
module Div20x (rst, clk, cet, cep, count,tc);
// TITLE 'Divide-by-20 Counter with enables'
// enable CEP is a clock enable only
// enable CET is a clock enable and
// enables the TC output
// a counter using the Verilog language

parameter size = 5;
parameter length = 20;

input rst; // These inputs/outputs represent
input clk; // connections to the module.
input cet;
input cep;

output [size-1:0] count;
output tc;

reg [size-1:0] count; // Signals assigned
                      // within an always
                      // (or initial)block
                      // must be of type reg

wire tc; // Other signals are of type wire

// The always statement below is a parallel
// execution statement that
// executes any time the signals
// rst or clk transition from low to high

always @ (posedge clk or posedge rst)
  if (rst) // This causes reset of the cntr
    count <= 5'b0;
  else
  if (cet && cep) // Enables both  true
    begin
      if (count == length-1)
        count <= 5'b0;
      else
        count <= count + 5'b1; // 5'b1 is 5 bits
    end                        // wide and equal
                               // to the value 1.

// the value of tc is continuously assigned
// the value of the expression
assign tc = (cet && (count == length-1));

endmodule
```

An example of delays:

```
...
reg a, b, c, d;
wire e;
...
always @(b or e)
 begin
   a = b & e;
   b = a | b;
   #5 c = b;
   d = #6 c ^ e;
 end
```

The always clause above illustrates the other type of method of use, i.e. the always clause executes any time any of the entities in the list change, i.e. the b or e change. When one of these changes, immediately a and b are assigned new values. After a delay of 5 time units, c is assigned the value of b and the value of c ^ e is tucked away in an invisible store. Then after 6 more time units, d is assigned the value that was tucked away.

Signals that are driven from within a process (an initial or always block) must be of type reg. Signals that are driven from outside a process must be of type wire. The keyword reg does not necessarily imply a hardware register.

Definition of constants

The definition of constants in Verilog supports the addition of a width parameter. The basic syntax is:

<Width in bits>'<base letter><number>

Examples:

- 12'h123 - Hexadecimal 123 (using 12 bits)
- 20'd44 - Decimal 44 (using 20 bits - 0 extension is automatic)
- 4'b1010 - Binary 1010 (using 4 bits)
- 6'o77 - Octal 77 (using 6 bits)

Synthesizeable constructs

There are several statements in Verilog that have no analog in real hardware, e.g. $display. Consequently, much of the language can not be used to describe hardware. The examples presented here are the classic subset of the language that has a direct mapping to real gates.

```
// Mux examples - Three ways to do the same thing.

// The first example uses continuous assignment
wire out ;
assign out = sel ? a : b;

// the second example uses a procedure
// to accomplish the same thing.

reg out;
always @(a or b or sel)
 begin
  case(sel)
   1'b0: out = b;
   1'b1: out = a;
  endcase
 end

// Finally - you can use if/else in a
// procedural structure.
reg out;
always @(a or b or sel)
  if (sel)
    out = a;
  else
    out = b;
```

The next interesting structure is a transparent latch; it will pass the input to the output when the gate signal is set for "pass-through", and captures the input and stores it upon transition of the gate signal to "hold". The output will remain stable regardless of the input signal while the gate is set to "hold". In the example below the "pass-through" level of the gate would be when the value of the if clause is true, i.e. gate = 1. This is read "if gate is true, the din is fed to latch_out continuously." Once the if clause is false, the last value at latch_out will remain and is independent of the value of din.

```
// Transparent latch example

reg out;
always @(gate or din)
 if(gate)
   out = din; // Pass through state
   // Note that the else isn't required here. The variable
   // out will follow the value of din while gate is high.
   // When gate goes low, out will remain constant.
```

The flip-flop is the next significant template; in Verilog, the D-flop is the simplest, and it can be modeled as:

```
reg q;
always @(posedge clk)
  q <= d;
```

The significant thing to notice in the example is the use of the non-blocking assignment. A basic rule of thumb is to use **<=** when there is a **posedge** or **negedge** statement within the always clause.

A variant of the D-flop is one with an asynchronous reset; there is a convention that the reset state will be the first if clause within the statement.

```
reg q;
always @(posedge clk or posedge reset)
  if(reset)
    q <= 0;
  else
    q <= d;
```

The next variant is including both an asynchronous reset and asynchronous set condition; again the convention comes into play, i.e. the reset term is followed by the set term.

```
reg q;
always @(posedge clk or posedge reset or posedge set)
 if(reset)
   q <= 0;
 else
 if(set)
   q <= 1;
 else
   q <= d;
```

Note: If this model is used to model a Set/Reset flip flop then simulation errors can result. Consider the following test sequence of events. 1) reset goes high 2) clk goes high 3) set goes high 4) clk goes high again 5) reset goes low followed by 6) set going low. Assume no setup and hold violations.

In this example the always @ statement would first execute when the rising edge of reset occurs which would place q to a value of 0. The next time the always block executes would be the rising edge of clk which again would keep q at a value of 0. The always block then executes when set goes high which because reset is high forces q to remain at 0. This condition may or may not be correct depending on the actual flip flop. However, this is not the main problem with this model. Notice that when reset goes low, that set is still high. In a real flip flop this will cause the output to go to a 1. However, in this model it will not occur because the always block is triggered by rising edges of set and reset - not levels. A different approach may be necessary for set/reset flip flops.

The final basic variant is one that implements a D-flop with a mux feeding its input. The mux has a d-input and feedback from the flop itself. This allows a gated load function.

```
// Basic structure with an EXPLICIT feedback path
always @(posedge clk)
  if(gate)
    q <= d;
  else
    q <= q; // explicit feedback path

// The more common structure ASSUMES the feedback is present
// This is a safe assumption since this is how the
// hardware compiler will interpret it. This structure
// looks much like a Latch. The differences are the
// '''@(posedge clk)''' and the non-blocking '''<='''
//
always @(posedge clk)
  if(gate)
    q <= d; // the "else" mux is "implied"
```

Initial and always

There are two separate ways of declaring a verilog process. These are the **always** and the **initial** keywords. The **always** keyword indicates a free-running process. The **initial** keyword indicates a process executes exactly once. Both constructs begin execution at simulator time 0, and both execute until the end of the block. Once an always block has reached its end, it is rescheduled (again). It is a common misconception to believe that an initial block will execute before an always block. In fact, it is better to think of the **initial**-block as a special-case of the **always**-block, one which terminates after it completes for the first time.

```
//Examples:
initial
  begin
    a = 1; // Assign a value to reg a at time 0
    #1; // Wait 1 time unit
    b = a; // Assign the value of reg a to reg b
  end

always @(a or b) // Any time a or b CHANGE, run the process
begin
  if (a)
    c = b;
  else
    d = ~b;
end // Done with this block, now return to the top (i.e. the @ event-
control)

always @(posedge a)// Run whenever reg a has a low to high change
  a <= b;
```

These are the classic uses for these two keywords, but there are two significant additional uses. The most common of these is an **always** keyword without the **@(...)** sensitivity list. It is possible to use always as shown below:

```
always
 begin // Always begins executing at time 0 and NEVER stops
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end // Keeps executing - so continue back at the top of the begin
```

The **always** keyword acts similar to the "C" construct **while(1) {..}** in the sense that it will execute forever.

The other interesting exception is the use of the **initial** keyword with the addition of the **forever** keyword.

The example below is functionally identical to the **always** example above.

```
initial forever // Start at time 0 and repeat the begin/end forever
 begin
   clk = 0; // Set clk to 0
   #1; // Wait for 1 time unit
   clk = 1; // Set clk to 1
   #1; // Wait 1 time unit
 end
```

Fork/join

The **fork/join** pair are used by Verilog to create parallel processes. All statements (or blocks) between a fork/join pair begin execution simultaneously upon execution flow hitting the **fork**. Execution continues after the **join** upon completion of the longest running statement or block between the **fork** and **join**.

```
initial
 fork
   $write("A"); // Print Char A
   $write("B"); // Print Char B
   begin
     #1; // Wait 1 time unit
     $write("C");// Print Char C
   end
 join
```

The way the above is written, it is possible to have either the sequences "ABC" or "BAC" print out. The order of simulation between the first $write and the second $write depends on the simulator implementation, and may purposefully be randomized by the simulator. This allows the simulation to contain both accidental race conditions as well as intentional non-deterministic behavior.

Notice that VHDL cannot dynamically spawn multiple processes like Verilog.

Race conditions

The order of execution isn't always guaranteed within Verilog. This can best be illustrated by a classic example. Consider the code snippet below:

```
initial
  a = 0;

initial
  b = a;

initial
  begin
    #1;
    $display("Value a=%b Value of b=%b",a,b);
  end
```

What will be printed out for the values of a and b? Depending on the order of execution of the initial blocks, it could be zero and zero, or alternately zero and some other arbitrary uninitialized value. The $display statement will always execute after both assignment blocks have completed, due to the #1 delay.

Operators

Note: these operators are NOT shown in order of precedence.

Operator type	**Operator symbols**	**Operation performed**
	~	1's complement
	&	Bitwise AND
Bitwise	|	Bitwise OR
	^	Bitwise XOR
	~^ or ^~	Bitwise XNOR
	!	NOT
Logical	&&	AND
	||	OR
	&	Reduction AND
	~&	Reduction NAND
Reduction	|	Reduction OR
	~|	Reduction NOR
	^	Reduction XOR
	~^ or ^~	Reduction XNOR
	+	Addition
	-	Subtraction
Arithmetic	-	2's complement
	*	Multiplication
	/	Division
	**	exponent (*Verilog-2001)
	>	Greater than
	<	Less than
	>=	Greater than or equal to
	<=	Less than or equal to
Relational	==	logical equality (bit-value 1'bX is removed from comparison)
	!=	Logical inequality (bit-value 1'bX is removed from comparison)
	===	4-state logical equality (bit-value 1'bX is taken as

		literal)
	!==	4-state Logical inequality (bit-value 1'bX is taken as literal)
Shift	>>	Logical Right shift
	<<	Logical Left shift
	>>>	Arithmetic Right shift (*Verilog-2001)
	<<<	Arithmetic Left shift (*Verilog-2001)
Concatenation	{ , }	Concatenation
Replication	{n{m}}	Replicate value m for n times
Conditional	? :	Conditional

System tasks

System tasks are available to handle simple I/O, and various design measurement functions. All system tasks are prefixed with **$** to distinguish them from user tasks and functions. This section presents a short list of the most often used tasks. It is by no means a comprehensive list.

- $display - Print to screen a line followed by an automatic newline.
- $write - Write to screen a line without the newline.
- $swrite - Print to variable a line without the newline.
- $sscanf - Read from variable a format-specified string. (*Verilog-2001)
- $fopen - Open a handle to a file (read or write)
- $fdisplay - Write to file a line followed by an automatic newline.
- $fwrite - Write to file a line without the newline.
- $fscanf - Read from file a format-specified string. (*Verilog-2001)
- $fclose - Close and release an open file-handle.
- $readmemh - Read hex file content into a memory array.
- $readmemb - Read binary file content into a memory array.
- $monitor - Print out all the listed variables when any change value.
- $time - Value of current simulation time.
- $dumpfile - Declare the VCD (Value Change Dump) format output file name.
- $dumpvars - Turn on and dump the variables.
- $dumpports - Turn on and dump the variables in Extended-VCD format.
- $random - Return a random value.

Program Language Interface (PLI)

The PLI provides a programmer with a mechanism to transfer control from Verilog to a program function written in C language. It is officially deprecated by IEEE Std 1364-2005 in favor of the newer Verilog Procedural Interface, which completely replaces the PLI.

The PLI enables Verilog to cooperate with other programs written in the C language such as test harnesses, instruction set simulators of a microcontroller, debuggers, and so on. For example, it provides the C functions `tf_putlongp()` and `tf_getlongp()` which are used to write and read the argument of the current Verilog task or function, respectively.