History of Programming Languages 16

Abstract

This chapter presents a short history of programming languages, starting with machine languages, to assembly languages, to early high-level procedural languages such as FORTRAN and COBOL, to later high-level languages such as Pascal and C and to object-oriented languages such as C++ and Java. Functional programming languages and logic programming languages are discussed, and there is a short discussion on the important area of syntax and semantics.

Key Topics

Generations of programming languages
Imperative languages
ALGOL
FORTRAN and COBOL
Pascal and C
Object-oriented languages
Java and C++
Functional programming languages
Logic programming languages
Syntax and semantics

16.1 Introduction

Hardware is physical and may be seen and touched, whereas software is intangible and is an intellectual undertaking by a team of programmers. Software is written in a particular programming language, and hundreds of languages have been

G. O'Regan, *Introduction to the History of Computing*, Undergraduate Topics in Computer Science, DOI 10.1007/978-3-319-33138-6_16

developed. Programming languages have evolved from the early days of computing with the earliest languages using machine code to instruct the computer. The next development was the use of assembly languages to represent machine language instructions. These were then translated into machine code by an assembler. The next step was to develop high-level programming languages such as FORTRAN and COBOL. These were easier to use than assembly languages and machine code and helped to improve quality and productivity.

A *first-generation* programming language (or 1GL) is a machine-level programming language that consists of 1 and 0 s. The main advantage of these languages is execution speed as they may be directly executed on the computer. These languages do not require a compiler or assembler to convert from a high-level language or assembly language into the machine code.

However, writing a program in machine code is difficult and error prone, as it involves writing a stream of binary numbers. This made the programming language difficult to learn and difficult to correct should any errors occur. The programming instructions were entered through the front panel switches of the computer system, and adding new code was difficult. Further, the machine code was not portable as the machine language for one computer could differ significantly from that of another computer. Often, the program needed to be totally rewritten for the new computer.

First-generation languages were used mainly on the early computers. A program written in a high-level programming language is generally translated by the compiler[1] into the machine language of the target computer for execution.

Second-generation languages, or 2GL, are low-level assembly languages that are specific to a particular computer and processor. However, assembly languages are easier to read than the first-generation machine code. They require considerably more programming effort than high-level programming languages and are more difficult to use for larger applications. The assembler converts the assembly code into the actual machine code to run on the computer. The assembly language is specific to a particular processor family and environment and is therefore not portable.

A program written in assembly language for a particular processor family needs to be rewritten for a different platform. However, since the assembly language is in the native language of the processor, it has significant speed advantages over high-level languages. Second-generation languages are still used today, but high-level programming languages have generally replaced them.

The *third-generation languages*, or 3GL, include high-level programming languages such as Pascal, C or FORTRAN. They are general-purpose languages and

[1] This is true of code generated by native compilers. Other compilers may compile the source code to the object code of a virtual machine, and the translator module of the virtual machine translates each byte code of the virtual machine to the corresponding native machine instruction. That is, the virtual machine translates each generalized machine instruction into a specific machine instruction (or instructions) that may then be executed by the processor on the target computer. Most computer languages such as C require a separate compiler for each computer platform (i.e. computer and operating system). However, a language such as Java comes with a virtual machine for each platform. This allows the source code statements in these programs to be compiled just once, and they will then run on any platform.

have been applied to business, scientific and general applications. They are designed to be easier for a human to understand and include features such as named variables, conditional statements, iterative statements, assignment statements and data structures. Early examples of third-generation languages are FORTRAN, ALGOL and COBOL. Later examples are languages such as C, C++ and Java. The advantages of these high-level languages are:

- Ease of readability
- Clearly defined syntax (and semantics[2])
- Suitable for business or scientific applications
- Machine independent
- Portability to other platforms
- Ease of debugging
- Execution speed

These languages are machine independent and may be compiled for different platforms. The early 3GLs were *procedural* in that they focus on how something is done rather than on what needs to be done. The later 3GLs were *object oriented*,[3] and the programming tasks were divided into objects. Objects may be employed to build larger programs, in a manner that is analogous to building a prefabricated building. Examples of modern object-oriented language are the Java language that is used to build web applications, C++ and Smalltalk.

High-level programming languages allow programmers to focus on problem-solving rather than on the low-level details associated with assembly languages. They are easier to debug and to maintain than assembly languages.

Fourth-generation languages specify what needs to be done rather than how it should be done. They are designed to reduce programming effort and include report generators and form generators. Report generators take a description of the data format and the report that is to be created and then automatically generate a program to produce the report. Form generators are used to generate programs to manage online interactions with the application system users. However, 4GLs are slow when compared to compiled languages.

A *fifth-generation* programming language, or 5GL, is a programming language that is based around solving problems using constraints applied to the program, rather than using an algorithm written by the programmer. Fifth-generation languages are designed to make the computer (rather than the programmer) solve the problem. The programmer specifies the problem and the constraints to be satisfied and is not concerned with the algorithm or implementation details. These languages are mainly used for research purposes especially in the field of artificial intelligence.

[2] The study of programming language semantics commenced in the 1960s. It includes work done by Hoare on axiomatic semantics, work done by Gordon Plotkin on operational semantics and work done by Scott and Strachey on denotational semantics.

[3] Norwegian Research originally developed object-oriented programming with their work on Simula 67 in the late 1960s.

Prolog is one of the best known fifth generation languages, and it is a logic programming language.

The task of deriving an efficient algorithm from a set of constraints for a particular problem is non-trivial, and to date this step has not been successfully automated. Fifth-generation languages are used mainly in academia.

16.2 Plankalkül

The earliest high-level programming language was Plankalkül developed by Konrad Zuse in 1946. It means 'Plan' and 'Kalkül' or, in other words, a calculus of programs. It is a relatively modern language for a language developed in 1946. There was no compiler for the language at the time, and it was only 50 years later that a compiler was finally developed for the language. The Free University of Berlin designed and developed a compiler in 2000, and the first Plankalkül program was run over 50 years after its conception.

The language employs data structures and Boolean algebra and includes a mechanism to define more powerful data structures. Zuse demonstrated that the Plankalkül language could be used to solve scientific and engineering problems, and he wrote several example programs including programs for sorting lists and searching a list for a particular entry. The main features of Plankalkül are:

- A high-level language.
- Fundamental data types are arrays and tuples of arrays.
- While construct for iteration.
- Conditionals are addressed using guarded commands.
- There is no GOTO statement.
- Programs are non-recursive functions.
- Type of a variable is specified when it is used.

The main constructs of the language are variable assignment, arithmetical and logical operations, guarded commands and while loops. There are also some list and set processing functions.

16.3 Imperative Programming Languages

Imperative programming is a programming style that describes computation in terms of a program state and statements that change the program state. The term *imperative* is a command to carry out a specific instruction or action. Similarly, imperative programming consists of a set of commands to be executed on the computer, and it is therefore concerned with *how* the program will be executed. The execution of an imperative command generally results in a change of state.

Imperative programming languages are quite distinct from *functional* and *logical programming languages*. Functional programming languages, like Miranda, have no global state, and programs consist of mathematical functions that have no side effects. In other words, there is no change of state, and the variable *x* will have the same value later in the program as it does earlier. Logical programming languages, like Prolog, define *what* is to be computed, rather than *how* the computation is to take place.

Most commercial programming languages are imperative languages, with interest in functional programming languages and relational programming languages being mainly academic. Imperative programs tend to be more difficult to reason about due to the change of state. Assembly languages and machine code are imperative languages.

High-level imperative languages use program variables and employ commands such as assignment statements, conditional commands, iterative commands and calls to procedures. An assignment statement performs an operation on information located in memory and stores the results in memory. The effect of an assignment statement is a change of the program state. A conditional statement allows a statement to be executed only if a specified condition is satisfied. Iterative statements allow a statement (or group of statements) to be executed a number of times.

High-level imperative languages allow the evaluation of complex expressions. These may consist of arithmetic operations and function evaluations, and the resulting value of the expression is assigned to memory.

FORTRAN was developed in the mid-1950s, and it was one of the earliest programming languages. ALGOL was developed in the late 1950s and 1960s, and it became a popular language for the expression of algorithms. COBOL was designed in the late 1950s as a programming language for business use. George Kemeny and Thomas Kurtz designed the BASIC (Beginner's All-purpose Symbolic Instruction Code) programming language in 1963. Niklaus Wirth developed Pascal in the early 1970s as a teaching language. Denis Ritchie at Bell Labs developed the C programming language in the early 1970s.

The Ada programming language was developed for the US military in the early 1980s. Object-oriented languages are imperative but include features to support objects. Bjarne Stroustrup designed C++ in 1985 as an object-oriented extension of the C language. Sun Microsystems released Java in 1996.

16.3.1 FORTRAN and COBOL

FORTRAN (FORmula TRANslator) was the first high-level programming language to be implemented. John Backus at IBM developed it in the mid-1950s, and the first compiler was available in 1957. The language includes named variables, complex expressions and subprograms. It was designed for scientific and engineering applications and remains the most important programming language for these domains. The main statements of the language include:

– Assignment statements (using the = symbol)
– IF statements
– GOTO statements
– DO loops

Fortran II was developed in 1958, and it introduced subprograms and functions to support procedural (or imperative) programming. Each procedure (or subroutine) contains computational steps to be carried out when it is called (at any point) during program execution. This could include calls by other procedures or by itself. However, recursion was not allowed until FORTRAN 90. FORTRAN 2003 provides support for object-oriented programming.

The basic types supported in FORTRAN include Boolean, integer and real. Support for double precision and complex numbers was added later. The language included relational operators for equality (.EQ.), less than (.LT.), and so on. FORTRAN is good at handling numbers and computation, and this is especially useful for mathematical and engineering problems. The following code (written in FORTRAN 77) gives a flavour of the language.

```
      PROGRAM HELLOWORLD
C     FORTRAN 77 SOURCE CODE COMMENTS FOR HELLOWORLD
        PRINT '(A)', 'HELLO WORLD'
        STOP
      END
```

FORTRAN remains a popular scientific programming language for application such as climate modelling, simulations of the solar system, modelling the trajectories of artificial satellites and simulation of automobile crash dynamics.

Fig. 16.1 Grace Murray and UNIVAC

It was initially weak at handling input and output, which was important to business computing. This led to the development of the COBOL programming language in the late 1950s.

The Common Business Oriented Language (COBOL) was the first business programming language, and it was introduced in 1959. Grace Murray Hopper[4] (Fig. 16.1) and a group of computer professionals called the Conference on Data Systems Languages (CODASYL) designed it with the objective of improving the readability of software source code. It has an English-like syntax designed to make it easy to learn the language. The only data types in the language were numbers and strings of text, and these may be grouped into arrays and records. The language is verbose:DIVIDE A BY B GIVING C REMAINDER D

COBOL was the first computer language whose use was mandated by the US Department of Defense. The language remains in use today, and there is an object-oriented version of the language.

16.3.2 ALGOL

ALGOL (ALGOrithmic Language) is a family of imperative programming languages, and it was originally developed in the mid-1950s and later revised in ALGOL 60 and ALGOL 68. It was designed to address some of the problems in FORTRAN, but it was not widely used. This may have been due to the refusal of IBM to support ALGOL and the dominance of IBM in the computing field.

A committee of American and European computer scientists designed the language, and it had a significant influence on later language design. ALGOL 60 [Nau:60] was the most popular member of the family, and Edsger Dijkstra developed an early ALGOL 60 compiler. John Backus and Peter Naur developed a method for describing the syntax of the ALGOL 58 programming language, which is known as Backus-Naur Form (or BNF).

ALGOL includes data structures and block structures. Block structures were designed to allow blocks of statements to be created (e.g. for procedures or functions). A variable defined within a block may be used within the block but is out of scope outside of the block.

ALGOL 60 introduced two ways of passing parameters to subprograms, and these are *call by value* and *call by name*. The call by value parameter passing technique involves the evaluation of the arguments of a function or procedure before the function or procedure is entered. The values of the arguments are passed to the function or procedure, and any changes to the arguments within the called function or procedure have no effect on the actual arguments. The call by name parameter passing technique is the default parameter passing technique in ALGOL 60. It involves re-evaluating the actual parameter expression each time the formal parameter is read. Call by name is used today in C/C++ macro expansion.

[4] Mary Hopper was a programmer on the Mark I, Mark II, Mark III and UNIVAC 1 computers. She was the technical advisor to the CODASYL committee.

ALGOL 60 includes conditional statements and iterative statements. It supports recursions: i.e. it allows a function or procedure to call itself. It includes:

- *Dynamic arrays.* These are arrays in which the subscript range is specified by variables.
- *Reserved words.* These are keywords that are not allowed to be used as identifiers by the programmer.
- *User-defined data types.* These allow the user to design their own data types.
- ALGOL uses bracketed statement blocks and it was the first language to use *begin-end* pairs for delimiting blocks.

ALGOL was used mainly by researchers in the United States and Europe. There was a lack of interest to its adoption by commercial companies due to the absence of standard input and output facilities in its description. ALGOL 60 became the standard for the publication of algorithms, and it had a major influence on later language development.

ALGOL evolved during the 1960s but not in the right direction. The ALGOL 68 committee decided on a very complex design rather than the simple and elegant ALGOL 60 specification. Tony Hoare remarked that:

ALGOL 60 was a great improvement on its successors.

16.3.3 Pascal and C

Niklaus Wirth designed the Pascal programming language in the early 1970s. It is named after Blaise Pascal (a seventeenth-century French mathematician), and it was based on the ALGOL programming language. It was intended as a language to teach students structured programming.

Structured programming [Dij:68] is concerned with rigorous techniques to design and develop programs, and there was intense debate on correct approaches to software development in the late 1960s. Dijkstra argued against the use of the GOTO statement 'GOTO Statement considered harmful' [Dij:68], and this influenced language design and led to several languages that did not include the GOTO statement.

The Pascal language includes constructs such as the conditional if statement; the iterative while, repeat and for statements; the assignment statement; and the case statement (which is a generalized if statement). The statement in the body of the repeat statement is executed at least once, whereas the statement within the body of a while statement may never be executed.

The language has several reserved words (known as keywords) that have a special meaning, and these may not be used as program identifiers. The Pascal program that displays 'Hello World' is given by:

```
program HELLOWORLD (OUTPUT);

begin
   WRITELN ('Hello, World!')
end.
```

Pascal includes several simple data types such as Boolean, integer, character and real. It also allows more advanced data types including arrays, enumeration types, ordinal types and pointer data types. It allows complex data types to be constructed from existing data types. Types are introduced by the reserved word 'type'.

```
type
  c = record
       a: integer;
       b: char
     end;
```

Pascal includes a 'pointer' data type, and this data type allows linked lists to be created by including a pointer type field in the record. The variable LINKLIST is a pointer to the data type B in the example below where B is a record:

```
type
 BPTR = ^B;
 B  = record
      A  : integer;
      C : BPTR
     end;
var
 LINKLIST : BPTR;
```

Pascal is a block-structured language with programs structured into procedures and function blocks. These can be nested to any depth, and recursion is allowed. Each block has its own constants, types, variables and other procedures and functions, which are defined, within the scope of the block.

Pascal was criticized as being unsuitable for serious programming by Brian Kernighan and others [Ker:81]. Many of these deficiencies were addressed in later versions of the language. However, by then Denis Richie at Bell Labs had developed the C programming language, which became popular in industry. It is a general-purpose and a systems programming language.

It was originally designed as a language to write the kernel for the UNIX operating system. This was novel as operating systems were traditionally written in assembly languages. The success of C in writing the UNIX kernel led to its use on several other operating systems such as Windows and Linux. It also influenced later language development such as C++, and it is one of the most commonly used system programming languages. The language is described in detail in [KeR:78].

The language provides high-level and low-level capabilities, and a C program that is written in ANSI C with portability in mind may be compiled for a very wide variety of computer platforms and operating systems with minimal changes to the source code. The C language is now available on a wide range of platforms.

C is a procedural programming language and includes conditional statements such as the 'if statement', the 'switch statement', iterative statements such as the 'while' statement or 'do' statement and the assignment statement.

- If statement
```
    if (A == B)
        A = A+1;
    else
        A=A - 1;
```
[5]
- Assignment statement
```
    i=i+1;
```

One of the first programs that people write in C is the Hello World program. This is given by:

```
main()

{
  printf("Hello, World\n");
}
```

It includes several predefined data types including integers and floating-point numbers.

- int (integer)
- long (long integer)
- float (floating-point real)
- double (double-precision real)

It allows more complex data types to be created using 'structs', which are similar to records in Pascal. It allows the use of pointers to access memory locations, which allows the memory locations to be directly referenced and modified. The result of the following example is to assign 5 to the variable x:

```
int  x;
int *ptr_x;

x=4;
ptr_x= &x;
*ptr_x =5;
```

C is a block-structured language, and a program is structured into functions (or blocks). Each function block contains its own variables and functions. A function may call itself (i.e. recursion is allowed).

[5]The semi-colon in Pascal is used as a statement separator, whereas it is used as a statement terminator in C.

One key criticism of C is that it is very easy to make errors in C programs and to thereby produce undesirable results. For example, one of the easiest mistakes to make is to accidentally write the assignment operator (=) for the equality operator (==). This totally changes the meaning of the original statement as can be seen below:

```
if (a == b)
    a++;        .... Program fragment A
else
    a--
if (a=b)
    a++;        .... Program fragment B
 else
    a--
```

Both program fragments are syntactically correct and the intended meaning of a program is easily changed. The philosophy of C is to allow statements to be written as concisely as possible, and this is potentially dangerous.[6] The use of pointers potentially leads to problems as uninitialized pointers may point anywhere in memory and may therefore write anywhere in memory. Therefore, the effective use of C requires experienced programmers, well-documented source code and formal peer reviews of the source code by other developers.

16.4 Object-Oriented Languages

The traditional view of programming is that a program is a collection of functions or a list of instructions to be performed on the computer. *Object-oriented programming* is a paradigm shift in programming, where a computer program is considered to be a collection of objects that act on each other. Each object is capable of sending and receiving messages and processing data. That is, each object may be viewed as an independent entity or actor with a distinct role or responsibility.

An object is a *black box* which sends and receives *messages*. A black box consists of *code* (computer instructions) and *data* (information which these instructions operate on). The traditional way of programming kept code and data separate. For example, functions and data structures in the C programming language are not connected. However, in the object-oriented world, code and data are merged into a single indivisible thing called an *object*.

The reason that an object is called a black box is that the user of an object never needs to look inside the box, since all communication to it is done via messages. Messages define the *interface* to the object. Everything an object can do is represented by its message interface. Therefore, there is no need to know anything about what is in the black box (or object) in order to use it. The access to an object is only

[6] It is very easy to write incomprehensible code in C and even one line of C code can be incomprehensible. The maintenance of poorly written code is a challenge unless programmers follow good programming practice. This discipline needs to be enforced by formal reviews of the source code.

Table 16.1 Object-oriented paradigm

Feature	Description
Class	A class defines the abstract characteristics of a thing, including its attributes (or properties), and its behaviours (or methods). The members of a class are termed objects
Object	An object is a particular instance of a class with its own set of attributes. The set of values of the attributes of a particular object is called its state
Method	The methods associated with a class represent the behaviours of the objects in the class
Message passing	Message passing is the process by which an object sends data to another object or asks the other object to invoke a method
Inheritance	A class may have subclasses (or children classes) that are more specialized versions of the class. A subclass inherits the attributes and methods of the parent class. This allows the programmer to create new classes from existing classes. The derived classes inherit the methods and data structures of the parent class
Encapsulation (information hiding)	One fundamental principle of the object-oriented world is encapsulation (or information hiding). The internals of an object are kept private to the object and may not be accessed from outside the object. That is, encapsulation hides the details of how a particular class works and it requires a clearly specified interface around the services provided
Abstraction	Abstraction simplifies complexity by modelling classes and removing all unnecessary detail. All essential detail is represented, and non-essential information is ignored.
Polymorphism	Polymorphism is behaviour that varies depending on the class in which the behaviour is invoked. Two or more classes may react differently to the same message. The same name is given to methods in different subclasses, i.e. one interface, and multiple methods

through its messages, while keeping the internal details private. This is called *information hiding*[7] and is due to work by Parnas in the early 1970s.

The origins of object-oriented programming go back to the invention of Simula 67 at the Norwegian Computing Research Centre[8] in the late 1960s. It introduced the notion of a class and instances of a class.[9] Simula 67 influenced later languages such as the Smalltalk object-oriented language developed at Xerox PARC in the mid-1970s. Xerox introduced the term *object-oriented programming* for the use of objects and messages as the basis for computation. Most modern programming languages support object-oriented programming (e.g. Java and C++), and object-oriented features are added to many existing languages such as BASIC, FORTRAN and Ada. The main features of object-oriented languages are described in Table 16.1.

[7] Information hiding is a key contribution by Parnas to computer science. He has also done work on mathematical approaches to software quality using tabular expressions [ORg:06].

[8] The inventors of Simula 67 were Ole-Johan Dahl and Kristen Nygaard.

[9] Dahl and Nygaard were working on ship simulations and were attempting to address the huge number of combinations of different attributes from different types of ships. Their insight was to group the different types of ships into different classes of objects, with each class of objects being responsible for defining its own data and behaviour.

Object-oriented programming has become popular in large-scale software development, and it became the dominant paradigm in programming from the early 1990s. Its proponents argue that it is easier to learn and simpler to develop and maintain such programs. Its growth in popularity was helped by the rise in popularity of graphical user interfaces (GUI), which is well suited to object-oriented programming. The C++ programming language has become popular, and it is an object-oriented extension of the C programming language.

16.4.1 C++ and Java

Bjarne Stroustrup developed the C++ programming language in 1983 as an object-oriented extension of the C programming language. It was designed to use the power of object-oriented programming and to maintain the speed and portability of C. It provides a significant extension of C's capabilities, but it does not force the programmer to use the object-oriented features of the language.

A key difference between C++ and C is the concept of a class. A *class* is an extension to the C concept of a structure. The main difference is that while a C data structure can hold only data, a C++ class may hold both data and functions. An *object* is an instantiation of a class: i.e. the class is essentially the type, whereas the object is essentially a variable of that type. Classes are defined in C++ by using the keyword class:

```
class class_name
{
    access_specifier_1:
       member1;
    access_specifier_2:
       member2;
    …
}
```

The members may be either data or function declarations, and an access specifier is included to specify the access rights for each member (e.g. private, public or protected). Private members of a class are accessible only by other members of the same class; public members are accessible from anywhere where the object is visible; protected members are accessible by other members of the same class and also from members of their derived classes. An example of a class in C++ is the definition of the class rectangle:

```
class CRectangle
{
  int x, y;
  public:
    void set_values (int,int);
    int area (void);
} rect;
```

Java is an *object-oriented programming language* developed by *James Gosling* and others at *Sun Microsystems* in the early 1990s. C and C++ influenced the syntax of the language, and the language was designed with portability in mind. The objective is for a program to be written once and executed anywhere. *Platform independence* is achieved by compiling the Java code into Java *bytecode, which* are simplified machine instructions specific to the Java platform.

This code is then run on a Java *virtual machine* (JVM) that interprets and executes the Java bytecode. The JVM is specific to the native code on the host hardware. The problem with interpreting bytecode is that it is slow compared to traditional compilation. However, Java has a number of techniques to address this including just in time compilation and dynamic recompilation. Java also provides automatic garbage collection. This is a very useful feature as it protects programmers who forget to deallocate memory (thereby causing memory leaks).

Java is a proprietary standard that is controlled through the Java Community Process. Sun Microsystems makes most of its Java implementations available without charge. The following is an example of the Hello World program written in Java:

```
class HelloWorld
{
  public static void main (String args[])
  {
      System.out.println ("Hello World!");
  }
}
```

16.5 Functional Programming Languages

Functional programming is quite distinct from imperative programming in that *it* involves the evaluation of *mathematical functions*. Imperative programming involves the execution of sequential (or iterative) commands that change the state. For example, the assignment statement alters the value of a variable, and the value of a given variable *x* may change during program execution.

There are no changes of state for functional programs. The fact that the value of *x* will always be the same makes it easier to reason about functional programs than imperative programs. Functional programming languages provide *referential transparency*: i.e. equals may be substituted for equals, and if two expressions have equal values, then one can be substituted for the other in any larger expression without affecting the result of the computation.

Functional programming languages use *higher-order* functions,[10] recursion, lazy and eager evaluation, monads[11] and *Hindley-Milner-type inference* systems.[12] These languages are mainly being used in academia, but there has been some industrial use, including the use of Erlang for concurrent applications in industry. Alonzo Church developed lambda calculus in the 1930s, and it provides an abstract framework for describing mathematical functions and their evaluation. It provides the foundation for functional programming languages. Church employed lambda calculus to prove that there is no solution to the *decision problem* for first-order arithmetic in 1936.

Lambda calculus uses transformation rules, and one of these rules is variable substitution. The original calculus developed by Church was untyped, but typed lambda calculi have since been developed. Any computable function can be expressed and evaluated using lambda calculus, but there is no general algorithm to determine whether two arbitrary lambda calculus expressions are equivalent. Lambda calculus influenced *functional programming languages* such as *LISP*, *ML* and *Haskell.*

Functional programming uses the notion of *higher-order functions*. Higher-order functions take other functions as arguments and may return functions as results. The derivative function $^{d}/_{dx} f(x) = f'(x)$ is a higher-order function. It takes a function as an argument and returns a function as a result. For example, the derivative of the function Sin(x) is given by Cos(x). Higher-order functions allow *currying which is* a technique developed by Schönfinkel. It allows a function with several arguments to be applied to each of its arguments one at a time, with each application returning a new (higher-order) function that accepts the next argument. This allows a function of n arguments to be treated as n applications of a function with one argument.

John McCarthy developed LISP at MIT in the late 1950s, and this language includes many of the features found in modern functional programming languages.[13] *Scheme* built upon the ideas in LISP. *Kenneth Iverson* developed *APL*[14] *in the early 1960s, and this language influenced Backus's FP programming language.* Robin Milner designed the ML programming language *in the early 1970s. David Turner* developed *Miranda in the mid-1980s*. The *Haskell programming language* was released in the late 1980s.

[10] Higher-order functions are functions that take functions as arguments or return a function as a result. They are known as operators (or functionals) in mathematics, and one example is the derivative function $^{dy}/_{dx}$ that takes a function as an argument and returns a function as a result.

[11] Monads are used in functional programming to express input and output operations without introducing side effects. The Haskell functional programming language makes use of uses this feature.

[12] This is the most common algorithm used to perform type inference. Type inference is concerned with determining the type of the value derived from the eventual evaluation of an expression.

[13] Lisp is a multi-paradigm language rather than a functional programming language.

[14] Iverson received the Turing Award in 1979 for his contributions to programming language and mathematical notation. The title of his Turing Award paper was 'Notation as a tool of thought'.

16.5.1 Miranda

Miranda was developed by David Turner at the University of Kent in the mid-1980s [Turn:85]. It is a non-strict functional programming language: i.e. the arguments to a function are not evaluated until they are actually required within the function being called. This is also known as lazy evaluation, and one of its main advantages is that it allows infinite data structures to be passed as an argument to a function. Miranda is a pure functional language in that there are no side effect features in the language. The language has been used for:

– Rapid prototyping
– Specification language
– Teaching language

A Miranda program is a collection of equations that define various functions and data structures. It is a strongly typed language with a terse notation.

```
z = sqr p / sqr q
       sqr k = k * k
       p = a + b
       q = a - b
       a = 10
       b = 5
```

The scope of a formal parameter (e.g. the parameter k above in the function sqr) is limited to the definition of the function in which it occurs.

One of the most common data structures used in Miranda is the list. The empty list is denoted by [], and an example of a list of integers is given by [1, 3, 4, 8]. Lists may be appended to by using the '++' operator. For example,

```
[1, 3, 5] ++ [2, 4] is [1, 3, 5, 2, 4].
```

The length of a list is given by the '#' operator:

```
# [1, 3] = 2
```

The infix operator ':' is employed to prefix an element to the front of a list. For example,

```
5 : [2, 4, 6] is equal to [5, 2, 4, 6]
```

The subscript operator '!' is employed for subscripting. For example,

```
Nums = [5,2,4,6]  then  Nums!0 is 5.
```

The elements of a list are required to be of the same type. A sequence of elements that contains mixed types is called a tuple. A tuple is written as follows:

```
Employee = (“Holmes”, “222 Baker St. London”, 50, “Detective”)
```

A tuple is similar to a record in Pascal whereas lists are similar to arrays. Tuples cannot be subscripted but their elements may be extracted by pattern matching. Pattern matching is illustrated by the well-known example of the factorial function:

```
fac 0 = 1

fac (n+1) = (n+1) * fac n
```

The definition of the factorial function uses two equations, distinguished by the use of different patterns in the formal parameters. Another example of pattern matching is the reverse function on lists:

```
reverse [] = []

reverse (a:x) = reverse x ++ [a]
```

Miranda is a higher-order language, and it allows functions to be passed as parameters and returned as results. Currying is allowed and this allows a function of *n* arguments to be treated as *n* applications of a function with one argument. Function application is left associative: i.e. f x y means (f x) y. That is, the result of applying the function *f* to *x* is a function, and this function is then applied to *y*. Every function with two or more arguments in Miranda is a higher-order function.

16.5.2 Lambda Calculus

Lambda calculus (λ-calculus) was designed by *Alonzo Church* in the 1930s to study computability. It is a formal system that may be used to study function definition, function application, parameter passing and recursion. It may be employed to define what a *computable function* is, and any computable function may be expressed and evaluated using the calculus.

The lambda calculus is equivalent to the *Turing machine* formalism. However, lambda calculus emphasizes the use of transformation rules, whereas Turing machines are concerned with computability on primitive machines. Lambda calculus consists of a small set of rules:

Alpha-conversion rule (α-conversion)[15]
Beta-reduction rule (β-reduction)[16]
Eta-conversion (η-conversion)[17]

[15] This essentially expresses that the names of bound variables is unimportant.

[16] This essentially expresses the idea of function application.

[17] This essentially expresses the idea that two functions are equal if and only if they give the same results for all arguments.

Every expression in the λ-calculus stands for a function with a single argument. The argument of the function is itself a function with a single argument and so on. The definition of a function is anonymous in the calculus. For example, the function that adds one to its argument is usually defined as $f(x) = x + 1$. However, in λ-calculus the function is defined as:

$$\lambda x.x + 1 \qquad \left(\text{or equivalently as } \lambda z.\, z + 1\right)$$

The name of the formal argument x is irrelevant and an equivalent definition of the function is $\lambda\, z.\ z + 1$. The evaluation of a function f with respect to an argument (e.g. 3) is usually expressed by $f(3)$. In λ-calculus this would be written as $(\lambda\, x.\ x + 1)$ 3, and this evaluates to $3 + 1 = 4$. Function application is *left associative*: i.e. $f\, x\, y = (f\, x)\, y$. A function of two variables is expressed in lambda calculus as a function of one argument, which returns a function of one argument. This is known as *currying* and has been discussed earlier. For example, the function $\mathrm{f}(x, y) = x + y$ is written as $\lambda\, x.\ \lambda\, y.\ x + y$. This is often abbreviated to $\lambda\, x\, y.\ x + y$.

λ-Calculus is a simple mathematical system and its syntax is defined as follows:

```
<exp>::=<identifier>     |
     λ<identifier>.<exp> | --abstraction
     <exp><exp>      | --application
     (<exp>)
   -- Syntax of Lambda Calculus --
```

λ-Calculus's four lines of syntax plus *conversion* rules are sufficient to define B*ooleans*, *integers*, *data structures* and computations on them. It inspired LISP and modern functional programming languages.

16.6 Logic Programming Languages

Logic programming languages describe what is to be done, rather than how it should be done. These languages are concerned with the statement of the problem to be solved, rather than how the problem will be solved.

These languages use mathematical logic as a tool in the statement of the problem definition. Logic is a useful tool in developing a body of knowledge (or theory), and it allows rigorous mathematical deduction to derive further truths from the existing set of truths. The theory is built up from a small set of axioms or postulates and rules of inference derive further truths logically.

The objective of logic programming is to employ mathematical logic to assist with computer programming. Many problems are naturally expressed as a theory, and the statement of a problem to be solved is often equivalent to determining if a new hypothesis is consistent with an existing theory. Logic provides a rigorous way to determine this, as it includes a rigorous process for conducting proof.

Computation in logic programming is essentially logical deduction, and logic programming languages use first-order[18] predicate calculus. It employs theorem proving to derive a desired truth from an initial set of axioms. These proofs are constructive[19] in the sense that an actual object that satisfies the constraints is produced, rather than a reliance on a theoretical existence theorem. Logic programming specifies the objects, the relationships between them and the constraints that must be satisfied for the problem.

- The set of objects involved in the computation
- The relationships that hold between the objects
- The constraints of the particular problem

The language interpreter decides how to satisfy the particular constraints. *Artificial intelligence* influenced the development of logic programming, and J*ohn McCarthy*[20] demonstrated that *mathematical logic* could be used for expressing knowledge. The first logic programming language was *Planner developed by Carl Hewitt at MIT in 1969. It uses a* procedural approach for knowledge representation rather than McCarthy's declarative approach.

The best-known logic programming languages is Prolog, which was developed in the early 1970s by *Alain Colmerauer* and *Robert Kowalski*. It stands for *pro*gramming in *log*ic. It is a goal-oriented language that is based on predicate logic. Prolog became an ISO standard in 1995. The language attempts to solve a goal by tackling the subgoals that the goal consists of:

```
goal :- subgoal₁, …, subgoalₙ.
```

That is, in order to prove a particular goal, it is sufficient to prove subgoal$_1$ through subgoal*n*. Each line of a Prolog program consists of a rule or a fact, and the language specifies what exists rather than how. The following program fragment has one rule and two facts:

```
grandmother(G,S) :- parent(P,S), mother(G,P).
mother(sarah, isaac).
parent(isaac, jacob).
```

[18] First-order logic allows quantification over objects but not functions or relations. Higher-order logics allow quantification of functions and relations.

[19] For example, the statement $\exists x$ such that $x = \sqrt{4}$ states that there is an x such that x is the square root of 4, and the constructive existence yields that the answer is that $x = 2$ or x - -2, i.e. constructive existence provides more the truth of the statement of existence, and an actual object satisfying the existence criteria is explicitly produced.

[20] John McCarthy received the Turing Award in 1971 for his contributions to artificial intelligence. He also developed the programming language LISP.

The first line in the program fragment is a rule that states that G is the grandmother of S if there is a parent P of S and G is the mother of P. The next two statements are facts stating that isaac is a parent of jacob, and that sarah is the mother of isaac. A particular goal clause is true if all of its subclauses are true:

$$\texttt{goalclause}(V_g)\ \texttt{:-}\ \texttt{clause}_1(V_1),..,\texttt{clause}_m(V_m)$$

A Horn clause consists of a goal clause and a set of clauses that must be proven separately. Prolog finds solutions by *unification:* i.e. by binding a variable to a value. For an implication to succeed, all goal variables V*g* on the left side of :- must find a solution by binding variables from the clauses which are activated on the right side. When all clauses are examined and all variables in V*g* are bound, the goal succeeds. But if a variable cannot be bound for a given clause, then that clause fails. Following the failure, Prolog *backtracks*, and this involves going back to the left to previous clauses to continue trying to unify with alternative bindings. Backtracking gives Prolog the ability to find multiple solutions to a given query or goal.

Most logic programming languages use a simple searching strategy to consider alternatives:

If a goal succeeds and there are more goals to achieve, then remember any untried alternatives and go on to the next goal.

If a goal is achieved and there are no more goals to achieve, then stop with success.

If a goal fails and there are alternative ways to solve it, then try the next one.

If a goal fails and there are no alternate ways to solve it, and there is a previous goal, then go back to the previous goal.

If a goal fails and there are no alternate ways to solve it, and no previous goal, then stop with failure.

Constraint programming is a programming paradigm where relations between variables can be stated in the form of constraints. Constraints specify the properties of the solution and differ from the imperative programming languages in that they do not specify the sequence of steps to execute.

16.7 Syntax and Semantics

There are two key parts to any programming language, namely, its syntax and semantics. The syntax is the grammar of the language, and a program needs to be syntactically correct with respect to its grammar. The semantics of the language is deeper and determines the meaning of what has been written by the programmer. The semantics of a language determines what a syntactically valid program will compute. A programming language is therefore given by:

```
Programming Language = Syntax + Semantics
```

The theory of the syntax of programming languages is well established, and Backus-Naur Form[21] (BNF) is employed to specify the grammar of languages. The grammar of a language may be input into a parser, which determines whether the program is syntactically correct. Chomsky[22] defined a hierarchy of grammars (regular, context-free, context sensitive). A BNF specification consists of a set of rules such as

```
<symbol>::=<expression with symbols>
```

where < symbol > is a *nonterminal* and the expression consists of sequences of symbols and/or sequences separated by the vertical bar '|' which indicates a choice. Symbols that never appear on a left side are called terminals. The partial definition of the syntax of various statements in a programming language is given below:

```
<loop statement> ::=<while loop>|<for loop>
<while loop> ::= while ()<statement>
<for loop> ::= for ()<statement>
 ::=<assignment statement> |<loop statement>
<assignment statement> ::=<variable> :=<expression>
```

The example above includes various nonterminals (<loop statement>, <while loop>, <for loop>, <condition>, <expression>, <statement>, <assignment statement> and < variable>). The terminals include 'while', 'for', ':=', '(' and ')'. The production rules for < condition > and < expression > are not included.

There are various types of grammars such as regular grammars, context-free grammars and context-sensitive grammars. A parser translates the grammar of a language into a parse table. Each type of grammar has its own parsing algorithm to determine whether a particular program is valid with respect to its grammar.

16.7.1 Programming Language Semantics

The formal semantics of a programming language is concerned with the meaning of programs. A program is written according to the rules of the language, and the compiler then checks that it is syntactically correct, and if so, it generates the equivalent machine code.[23]

The compiler must preserve the semantics of the language, and the syntax of the language gives no information as to the meaning of a program. It is possible to write syntactically correct programs that behave in quite a different way from the intentions of the programmer.

[21] Backus-Naur Form is named after John Backus and Peter Naur. It was created as part of the design of ALGOL 60 and used to define the syntax rules of the language.

[22] Chomsky made important contributions to linguistics and the theory of grammars. He is more widely known today as a critic of US foreign policy.

[23] Of course, what the programmer has written may not be what the programmer had intended.

Table 16.2 Programming language semantics

Approach	Description
Axiomatic semantics	Axiomatic semantics involves giving meaning to phrases of the language with logical axioms. This approach is based on mathematical logic, and it employs pre and post condition assertions to specify what happens when the statement executes. The relationship between the initial assertion and the final assertion essentially gives the semantics of the code
Operational semantics	The operational semantics for a programming language was developed by Gordon Plotkin [Plo:81]. It describes how a valid program is interpreted by a sequence of computational steps
	An abstract machine (SECD machine) may be defined to give meaning to phrases, by describing the transitions they induce on states of the machine
	A precise mathematical interpreter (such as the lambda calculus) may also give the semantics
Denotational semantics	Denotational semantics (originally called mathematical semantics) provides meaning to programs in terms of mathematical objects such as integers, tuples and functions
	Each phrase in the language is translated into a mathematical object that is the *denotation* of the phrase. Christopher Strachey and Dana Scott developed it in the mid-1960s

The formal semantics of a language is given by a mathematical model, which describes the possible computations described by the language. The three main approaches to programming language semantic are axiomatic semantics, operational semantics and denotational semantics. A short summary of each approach is described in Table 16.2, and more detailed information is in [ORg:06, ORg:12].

16.8 Review Questions

1. Describe the five generations of programming languages.
2. Describe the early use of machine code.
3. Describe the early use of assembly languages.
4. Describe the key features of Fortran and COBOL.
5. Describe the key features of Pascal and C.
6. Discuss the key features of object-oriented languages.
7. Explain the differences between imperative programming languages and functional programming languages.
8. What are the key features of logic programming languages?
9. What is the difference between syntax and semantics?
10. Explain the main approaches to programming language semantics.

16.9 Summary

This chapter considered the evolution of programming languages from the older machine languages, to the low-level assembly languages, to high-level programming languages and object-oriented languages, to functional and logic programming languages. Finally, the syntax and semantics of programming languages were briefly discussed.

The advantages of the machine languages are execution speed and efficiency. It is difficult to write programs in these languages, as the program involves a stream of binary numbers. These languages were not portable, as the machine language for one computer may differ significantly from the machine language of another.

The second-generation languages, or 2GLs, are low-level assembly languages that are specific to a particular computer and processor. These are easier to write and understand, but they must be converted into the actual machine code to run on the computer. The assembly language is specific to a particular processor family and environment and is therefore not portable. However, their advantages are execution speed, as the assembly language is the native language of the processor.

The third-generation languages, or 3GLs, are high-level programming languages. They are general-purpose languages and have been applied to business, scientific and general applications. They are designed to be easier to understand and to allow the programmer to focus on problem solving. Their advantages include ease of readability and portability and ease of debugging and maintenance. The early 3GLs were procedure oriented and the later 3GLs were object oriented.

Fourth-generation languages, or 4GLs, are languages that consist of statements similar to human language. Most fourth-generation languages are non-procedural and are often used in database programming. They specify what needs to be done rather than how it should be done, and they have been used as report generators and form generators.

Fifth-generation programming languages or 5GLs, are programming languages that is based around solving problems using logic programming or applying constraints to the program. They are designed to make the computer (rather than the programmer) solve the problem. The programmer only needs to be concerned with the specification of the problem and the constraints to be satisfied and does not need to be concerned with the algorithm or implementation details.