T

Teaching Computer Languages in Universities

Kevin R. Parker[1], David V. Beard[1] and Bill Davey[2]
[1]Department of Informatics and Computer Science, Idaho State University, Pocatello, ID, USA
[2]School of Business IT and Logistics, RMIT University, Melbourne, VIC, Australia

Synonyms

Algorithmic language; Computer programming; Computer-oriented language; Computing language; Programming language; Software development

Definition

A computer language may refer to any of a variety of languages used to express a set of detailed instructions to a computer so that it performs specific tasks.

Introduction

This section will discuss a variety of issues surrounding the use of computers in universities and throughout higher education. The initial entries focus on teaching university students how to write computer programs and solve problems using algorithms.

This entry introduces computer languages, defined above as any of a variety of languages used to express a set of detailed instructions to a computer so that it performs specific tasks. Such languages are based on specific syntactic and semantic rules used to define the meaning of the programming language constructs.

Programming Overview

Various programming paradigms have emerged over time. A programming paradigm is a collection of coherent, often ideologically or theoretically based abstractions that together mold the design process and ultimately determine a program's structure (Wampler and Clark 2010; Ambler et al. 1992).

The paradigm being followed influences the program development process. There are various approaches for developing programs, including the process-oriented approach and the object-oriented approach.

Process-Oriented Approach

When using the process-oriented approach, the problem being analyzed is considered from the aspect of what processes or procedures must be performed in order to convert the provided input into the desired output. When following

A. Tatnall (ed.), *Encyclopedia of Education and Information Technologies*,
https://doi.org/10.1007/978-3-319-60013-0_49-1

this approach, each program specifies a list of instructions indicating how the program's purpose is accomplished. Developing a sequence of instructions for a computer to follow involves a certain process. The problem-solving phase involves analyzing the problem and devising a general solution called an algorithm. An algorithm is simply a step-by-step outline detailing how to solve a problem in a finite amount of time. After testing the algorithm to confirm that it works correctly, it can be translated into a programming language.

As programs become more complex, understandability can be improved if each process is considered separately. This is done by dividing the program into functions or subprograms. Each subprogram should have a clearly defined purpose and a clearly defined interface to the other subprograms in the program. This subdivision of programs can be carried further by grouping related subprograms into modules, referred to as modular design. The "divide and conquer" approach is one of the cornerstones of structured programming. Structured programming, along with its familiar features such as modular design, has long been a reliable approach to program development.

Object-Oriented Approach

One of the primary shortcomings of traditional program development techniques is their inability to easily handle the complexity associated with today's larger programs. Object-oriented (OO) development techniques are better able to cope with this complexity by offering several improved features such as data hiding, encapsulation, and polymorphism that make large programs easier to maintain and modify.

OO designs are thought to more accurately model the real world. Instead of viewing the world as a set of processes, it views the world as a collection of entities that are characterized by certain behaviors. Furthermore, OO languages make it easier to create new data types to better represent real-world objects.

The fundamental idea behind OO languages is to combine both data and the functions that operate on that data into a single unit called an object. In order to conceptualize a programming problem, the programmer determines what parts of the problem can be most usefully represented as objects, and then all the data and functions connected with that object are combined in the class specification. An object's functions, referred to as methods, provide the only way to access its data. In order to determine the value of a data item in an object, one of the object's methods must be called to read the data item and return its value. The data cannot be accessed directly. It is hidden and is safe from accidental alteration. Data and functions are said to be encapsulated into a single entity. Both data encapsulation and data hiding are key concepts in the object-oriented paradigm. Restricting changes to an object's data to its member functions ensures that no other functions can alter that data and therefore simplifies writing, debugging, and maintaining the program.

Language History

Both programming languages and program development techniques emerged near the middle of the twentieth century. Von Neumann's (1945) computer architecture led to very early programming languages, which were individual machine languages designed to control specific central processing units. UNIVAC's C-10 language, developed in 1949, was the first computer language and made use of mnemonic instructions, like "a" for add and "b" for bring. It was not long before high-level programming languages with more natural syntax became available to simplify computer programming. By the late 1950's, universities had discovered, and in many cases created, higher-level languages.

The Computer History Museum (http://www.computerhistory.org/) provides a visual timeline for the creation of early languages, which is summarized in Table 1.

FORTRAN, designed to ease the translation of mathematical formulas into code, was released in 1957. ALGOL, released in 1958 and updated in 1960, introduced recursion, indirect addressing, and character manipulation, among other features. Many universities adopted ALGOL for use in

Teaching Computer Languages in Universities, Table 1 Timeline of early language creation (from http://www.computerhistory.org/timeline/software-languages/)

Year	Development
1945	Zuse begins Plankalkül (plan calculus), the first algorithmic programming language
1948	Shannon identifies the bit as the fundamental unit of data and shows how to code data for transmission
1952	Hopper completes the A-0 compiler
1953	Backus creates Speedcode for the IBM 701
1957	IBM team led by Backus develops FORTRAN
1958	Algol 58, intended to be a universal language, released
1959	LISP created by McCarthy
1959	COBOL created by a team representing manufacturers and pentagon
1962	Iverson develops APL
1963	First edition of the ASCII standard published
1964	BASIC created at Dartmouth by Kurtz and Kemeny
1965	Simula written by Nygaard and Dahl
1969	UNIX developed at AT&T by Thompson and Ritchie
1970	Pascal programming language introduced by Wirth
1972	C programming language released by Thompson and Ritchie

their computer programming courses because it provided a precise and useful way for expressing algorithms (Keet 2004). COBOL was developed in 1959, and 80% of the world's daily business transactions still rely on COBOL (Beach 2014). Along with BASIC, these early higher-level imperative paradigm programming languages (Wilson and Clark 1993) used global variables, assignment statements, labeled commons, and goto statements to implement increasingly complex programs. While these early languages had subroutines, lengthy programs often featured multiple goto statements that branched throughout the rest of the code. The resulting "spaghetti code" (Conway 1978) was difficult to understand, debug, maintain, and extend (Dijkstra 1968).

In parallel, an alternate paradigm of functional programming languages evolved out of mathematical logic and recursion, with languages such as LISP released in 1959 (Turner 2012). LISP used many small isolated functions with if expressions and recursion instead of loops, allowing small portions of the overall problem to be implemented quickly and independently. Ideally, each function was implemented without any internal variables and with a single if expression similar to a mathematical formula. Each function was "side-effect free," meaning that there were no variables or structures outside the function that were affected by actions inside the function.

While functional programming languages such as LISP were used mainly in universities, the underlying concepts of small isolated independent functions and hierarchical decomposition were included in the paradigm of structured programming languages (Dijkstra 1972; Knuth 1974) and embraced by industry (Yourdon and Constantine 1975). Software developers were encouraged to use hierarchical decomposition and structure charts (Yourdon and Constantine 1975) to create many small procedures (functions and subroutines) with no global variables. Each of these procedures performed a single task, could be tested independently of the rest of the code, and could be reused in other parts of the code while avoiding global variables. Pass-by-value parameters were used whenever possible to further isolate these routines. Procedural or structured programming languages of this period typically included PL/I, ALGOL, and C.

As early as 1960 there were over 70 languages in existence, and by 1971 there were 164 (Sammet 1972). Sammet identifies the period from 1960 to 1970 as the decade in which the programming

language field matured. Higher-level languages became more widely used than machine-level coding because they made it possible to specify a problem solution in terms closer to human language.

By 1972, most universities in the USA and Australia had established computer science or data processing (which later evolved into information systems) degree programs. The majority of the computer science degree programs offered ALGOL, FORTRAN, or LISP, while most data processing programs offered COBOL. In Britain, BASIC was also widely used. During the late 1960s, departments experimented with a variety of other languages like PL/I.

The mid-1970s brought about the introduction of the microcomputer. These machines came with BASIC installed, and while that revolutionized the teaching of computer programming courses in high schools, the trend did not widely impact university programs. Instead, in the 1970s many universities adopted Pascal for use in their introductory programming course. Even as late as 1996, a survey of programs accredited by the Computing Sciences Accreditation Board found that 36% of the responding institutions listed Pascal as their first language (McCauley and Manaris 1998). The use of Pascal in academia was eventually supplanted by other languages beginning with C and C++ and eventually shifting to Java and C#.

The 1980s continued to experience an increase in the number of available languages. Tharp (1982) reviewed several comparison studies of programming languages including FORTRAN, COBOL, JOVIAL, Ada, ALGOL, Pascal, PL/I, and SPITBOL on the basis of their support of good software engineering practices, availability of control structures, programmer time required for developing a representative non-numeric algorithm, and the machine resources expended in compiling and executing it.

The structured programming paradigm began to give way to the OO paradigm in the early to mid-1990s as languages supporting OO became widely available. The emergence of the OO paradigm accelerated more extensive use of event-driven programming. In the event-driven programming paradigm, program execution flow is controlled not by pre-defined software constructs but rather by "events" that might be generated by a user clicking a mouse, an external event from the network, or by a timer.

Additional programming paradigms have emerged over time, including relational, functional, constraint-based, theorem-proving, concurrent, imperative, declarative, graphical, reflective, context-aware, rule-based, and agent-oriented (Wampler and Clark 2010). Today's applications, however, are seldom homogeneous and combine aspects from several different paradigms. Many complex systems consist of a variety of subcomponents that require a mixture of technologies. Therefore, using a single programming language and paradigm is becoming less common, with a movement toward multi-paradigm programming in which the heterogeneous subcomponents are each implemented with the appropriate paradigm. This may be accomplished through the use of multiple languages, an approach referred to as polyglot ("many tongues") programming (Wampler and Clark 2010).

Multiparadigm programming can also be achieved using a programming language designed to support multiple paradigms such as C++, which supports features from multiple paradigms including classes, overloaded functions, templates, modules, ordinary procedural programming, and macros (Coplien 1999). There are also various experimental programming languages that combine multiple paradigms.

Observations

This entry has provided a brief introduction to computing languages. Subsequent entries will discuss the selection of programming languages for higher education courses, followed by teaching software design techniques in university courses, comparative languages, teaching mobile computing, informatics education, health informatics education, approaches for teaching agile methodologies, and teaching robotics courses.

Cross-References

- Programming and Coding in Secondary Schools
- Programming Language Selection for University Courses
- Programming Languages for Secondary Schools
- Programming Languages for Secondary Schools, Java
- Programming Languages for Secondary Schools, Pascal
- Programming Languages for Secondary Schools, Python
- Programming Languages for University Courses

References

Ambler AL, Burnett MM, Zimmermann BA (1992) Operational versus definitional: a perspective on programming paradigms. IEEE Comput 25(9):28–43. Retrieved November 11, 2017 from ftp://ftp.cs.orst.edu/pub/burnett/Computer-paradigms-1992.pdf

Beach G (2014) Cobol is dead. Long live Cobol! Wall Street J. Retrieved November 11, 2017 from https://blogs.wsj.com/cio/2014/10/02/cobol-is-dead-long-live-cobol/

Conway R (1978) A primer on disciplined programming using PL/I, PL/CS, and PL/CT. Winthrop Publishers, Cambridge, MA

Coplien JO (1999) Multi-paradigm design for C++. Addison Wesley, Boston. Retrieved November 11, 2017 from http://www.inkdrop.net/docs/multiParadigm.pdf

Dijkstra EW (1968) Letters to the editor: go to statement considered harmful. Commun ACM 11(3):147–148. Retrieved November 11, 2017 from http://codeblab.com/wp-content/uploads/2009/12/Go-To-Statement.pdf

Dijkstra EW (1972) Notes on structured programming. In: Structured programming. Academic, London, pp 1–82. Retrieved November 11, 2017 from https://www.cs.utexas.edu/users/EWD/ewd02xx/EWD249.PDF

Keet EE (2004) A personal recollection of software's early days (1960–1979): part 1. IEEE Ann Hist Comput 26:46–61

Knuth D (1974) Structured programming with go to statements. Comput Surv 6(4):261–301. Retrieved November 11, 2017 from https://pic.plover.com/knuth-GOTO.pdf

McCauley R, Manaris B (1998) Computer science programs: what do they look like? A report on the annual survey of accredited programs. Proceedings of the 29th SIGCSE technical symposium on computer science education, pp 15–19. Retrieved November 11, 2017 from http://www.academia.edu/15180415/Computer_science_degree_programs_what_do_they_look_like_A_report_on_the_annual_survey_of_accredited_programs

Sammet JE (1972) Programming languages: history and future. Commun ACM 15(7):601. Retrieved November 11, 2017 from https://pdfs.semanticscholar.org/48af/15cfd104f7a5d91fef8f3136fe88502ada95.pdf

Tharp AL (1982) Selecting the "right" programming language. SIGCSE '82 Proceedings of the thirteenth SIGCSE technical symposium on computer science education, pp 151–155

Turner D (2012) Some history of functional programming languages. TFP 2012 Proceedings of the 2012 conference on trends in functional programming, vol 7829, pp 1–20. Retrieved November 11, 2017 from https://www.cs.kent.ac.uk/people/staff/dat/tfp12/tfp12.pdf

von Neumann J (1945) First draft of a report on the EDVAC, Contract No. W-670-ORD-4926, U.S. Army Ordnance Department, University of Pennsylvania Moore School of Electrical Engineering, Philadelphia. Retrieved November 12, 2017 from http://www.virtualtravelog.net/wp/wp-content/media/2003-08-TheFirstDraft.pdf

Wampler D, Clark T (2010) Multiparadigm programming. IEEE Softw 27(5):2–7. Retrieved November 11, 2017 from https://www.computer.org/csdl/mags/so/2010/05/mso2010050020.pdf

Wilson LB, Clark RG (1993) Comparative programming languages. Addison Wesley, Boston

Yourdon E, Constantine LL (1975) Structured design: fundamentals of a discipline of computer program and systems design. Yourdon Press, New York