Since P might have several choices available to it on each move, this simulation could take exponentially more time than P does. This might suggest that the task of parsing a context-free language can be prohibitively time-consuming, but in fact it is not. General-purpose, context-free parsing algorithms can be designed to require only time n^3, where n is the length of the input, by using *dynamic programming* (*see* ALGORITHMS, DESIGN AND CLASSIFICATION OF). One of the most popular such algorithms is *Earley's algorithm*. It takes time n^3 in the worst case, but for many context-free grammars it takes only a linear amount of time. The n^3 bound for an all-purpose, context-free parser can be improved slightly, but it is not yet known how much improvement is possible.

A non-deterministic pushdown automaton is a theoretical construct that is time-consuming to simulate in the real world. So, in searching for classes of context-free languages that are easy to parse, it is reasonable to consider *deterministic* context-free languages—those languages that can be recognized by a deterministic pushdown automaton. As one might expect, all deterministic context-free languages can be parsed rapidly; in fact, in a linear amount of time. But not all context-free languages are deterministic. For example, the set of all binary strings (strings of 0s and 1s) that are palindromes is context-free but not deterministic because a pushdown automaton for this language must of necessity operate something like this: by *guessing* when half the input has been read, store this portion of the string on the stack, and use the stack to verify that the second half of the input agrees symbol by symbol, in reverse order, with the first half.

So, non-deterministic pushdown acceptors are more powerful than deterministic ones. Are the corresponding statements true for the other kinds of automata used to characterize the families of languages in the Chomsky hierarchy? For finite-state automata and for Turing machines, the answer is no. It is easy to show that the non-deterministic versions of these devices are no more powerful than the deterministic versions. In other words, the ability to make guesses may enable these devices to do their jobs more quickly, but it will not let them do anything that they could not have done without guessing. But for linear-bounded automata, it is still not known whether the non-deterministic version (which corresponds to the context-sensitive languages) is more powerful than the deterministic version.

This question, called the *lba* problem, can be recast in the following form: can a Turing machine that performs a computation with the aid of guessing (i.e. of non-determinism), using just a linear amount of storage space, always be simulated by a comparably efficient Turing machine that does not need to guess? The analogous question for Turing machines that use a polynomially bounded amount of computation time rather than a linear amount of storage space is the very important P = NP problem (*see* COMPUTATIONAL COMPLEXITY and NP-COMPLETE PROBLEMS). In both cases, the answer is thought to be no, but such questions are notoriously difficult and have so far resisted all efforts at solution.

Bibliography

1972. Aho, A. V., and Ullman, J. D. *The Theory of Parsing, Translation and Compiling*. Upper Saddle River, NJ: Prentice Hall.

1973. Salomaa, A. *Formal Languages*. New York: Academic Press.

1978. Harrison, M. A. *Introduction to Formal Language Theory*. Reading, MA: Addison-Wesley.

1979. Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley.

1988. Moll, R. N., Arbib, M. A., and Kfoury, A. J. *An Introduction to Formal Language Theory*. New York: Springer-Verlag.

1996. Linz, P. *Introduction to Formal Languages and Automata*. Sudbury, MA: Jones and Bartlett.

Jonathan Goldstine

FORMAL METHODS FOR COMPUTER SYSTEMS

For articles on related subjects *see* AUTOMATIC PROGRAMMING; HARDWARE VERIFICATION; MODEL CHECKING; PROGRAM SPECIFICATION; and PROGRAM VERIFICATION.

Formal methods used in developing and verifying software and hardware systems are mathematically based techniques for describing and reasoning about system properties. Such formal methods provide frameworks within which people specify, develop, and verify systems in a systematic, rather than *ad hoc*, manner. Formal methods include the more specific activities of program specification, program verification, and hardware verification.

A method is formal if it has a sound mathematical basis, typically given by a *formal specification language*. This basis provides the means of precisely defining notions like consistency and completeness and, more relevantly, specification, implementation, and correctness. It provides the means of proving that a specification is realizable, proving that a system has been implemented correctly and proving properties of a system without necessarily running it to determine its behavior.

A formal method also addresses a number of pragmatic considerations: who uses it, what it is used for, when it is used, and how it is used. Most commonly, system

designers use formal methods to specify or verify a system's desired behavioral and structural properties. However, anyone involved in any stage of system development can make use of formal methods. They can be used in the initial statement of a customer's requirements, through system design, implementation, software testing (*q.v.*), debugging (*q.v.*), software maintenance (*q.v.*), program verification, and evaluation.

Formal methods are used to reveal ambiguity, incompleteness, and inconsistency in a system. When used early in the system development process, they can reveal design flaws that otherwise might be discovered only during costly testing and debugging phases. When used later (e.g. in verification), they can help determine the correctness of a system implementation and the equivalence of different implementations.

For a method to be formal, it need not address any pragmatic considerations, but a lack of such considerations would render it useless. Hence a formal method should possess a set of guidelines or a "style sheet" that tells the user the circumstances under which the method can and should be applied, as well as how it can be applied most effectively.

One tangible product of applying a formal method is a *formal specification*. A specification serves as a contract, a valuable piece of documentation, and a means of communication between a client, a specifier, and an implementer. Because of their mathematical basis, formal specifications are more precise and usually more concise than informal ones.

Since a formal method is a method and not just a computer program or language, it may or may not have software tools to support it. If the syntax of a formal method's specification language is made explicit, providing standard syntax analysis tools for formal specifications would be appropriate. If the language's semantics are sufficiently restricted, varying degrees of semantic analysis can be performed with machine aids as well. For example, under certain circumstances in hardware verification, the process of proving the correctness of an implementation against a specification can be completely automated. Thus, formal specifications have the additional advantage over informal ones of being amenable to machine analysis and manipulation.

Tremendous progress has been made in formal methods in the past few years. The most prominent successes have been in three areas: specification of large, complex software systems; the use of model checking for hardware and protocol verification; and the development of sophisticated theorem provers.

For software specification, the most common formal notations used today are Z, VDM, Larch, CSP, CCS, Statecharts, RAISE, LOTOS, temporal logic, and their variants. Examples of nontrivial case studies can be found in a wide range of application domains: avionics, databases, household electricity meters, medical devices, nuclear reactors, oscilloscopes, railways, security, and telephony.

Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The most common model checkers used today are SMV, Caesar, the Concurrency Workbench, Spin, Murϕ, and Cospan/FormalCheck. Model checkers have been most successfully used to verify and debug hardware designs and cache coherence and communication protocols.

Theorem proving (*q.v.*) is a process of finding a proof of a property of a system where both the system and the property are expressed as formulas in some mathematical logic. Examples of sophisticated theorem provers and proof checkers include PVS, ACL2, Nqthm, STeP, LP, HOL, and Coq. Recent efforts in using these systems have focused at the hardware level, e.g. verifying processor designs, microcode, or instruction sets.

For a comprehensive survey of the state of the art, including descriptions of and citations to numerous case studies, *see* Clarke and Wing (1996). For more on the benefits of formal specification, *see* Meyer (1985) and Hinchey and Bowen (1995). For a gentle introduction to formal methods, including simple examples in common specification notations, *see* Wing (1990).

Bibliography

1985. Meyer, B. "On Formalism in Specification," *IEEE Software*, **2**, 6–26.

1990. Wing, J. "A Specifier's Introduction to Formal Methods," *IEEE Computer*, **23**, *9*, 8–24.

1995. Hinchey, M. G., and Bowen, J. P. (eds.) *Applications of Formal Methods*. Upper Saddle River, NJ: Prentice Hall.

1996. Clarke, E. M., and Wing, J. M. "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys*, **28**, *4*, 626–643.

1999. Bowen, J. "Formal Methods" (part of the Virtual Library). `http://www.comlab.ox.ac.uk/archive/formal-methods.html`.

Jeannette M. Wing

FORTH

For articles on related subjects *see* EXTENSIBLE LANGUAGE; POLISH NOTATION; PROGRAMMING LANGUAGES; REAL-TIME SYSTEMS; and STACK.

History

Forth is a programming language and environment invented by Charles H. Moore in 1970. It was designed