Aho & Ullman

An Eternal Golden Braid

Christiano Braga - June 23, 2021

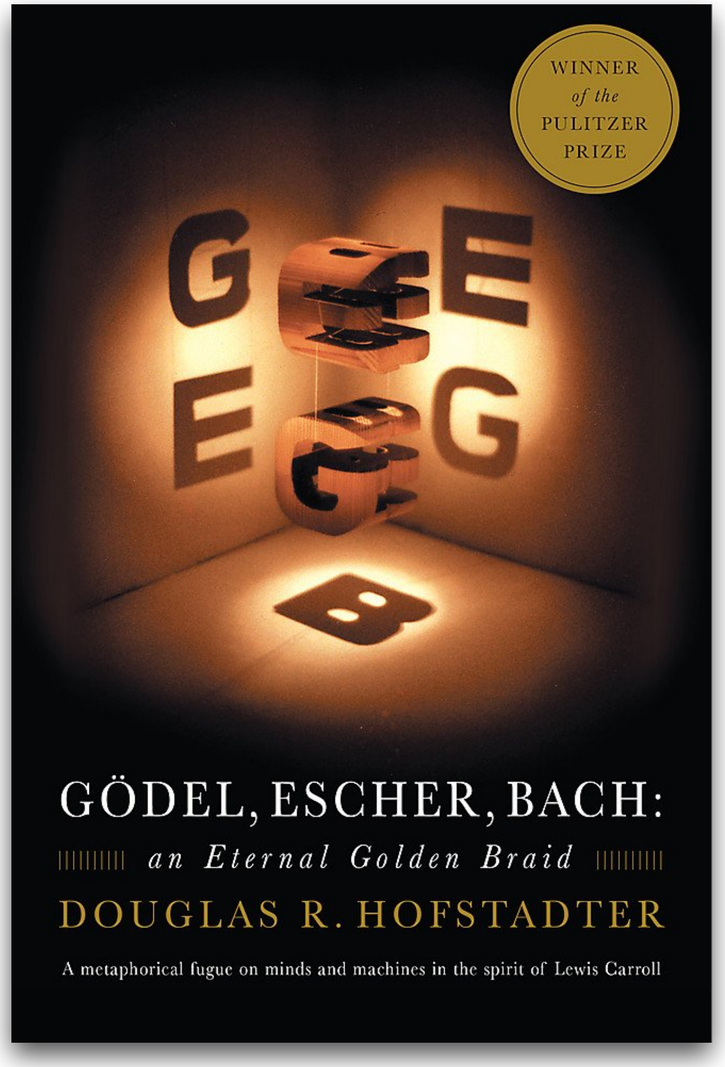
Who am I?

Christiano Braga

- Associate Professor at Universidade Federal Fluminense
- D.Sc. in Informatics, PUC-Rio, 2001 (with a 18 months internship at SRI International)
- Research interests include formal semantics of programming languages and formal compiler construction

June 2021 CACM: 2020 ACM A.M. Turing Award

Why the title?



2020 A.M. Turing Award

ACM Turing Award Honors Innovators Who Shaped the Foundations of Programming Language Compilers and Algorithms

Columbia's Aho and Stanford's Ullman Developed Tools and Fundamental Textbooks Used by Millions of Software Programmers around the World

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

Computer software powers almost every piece of technology with which we interact. Virtually every program running our world—from those on our phones or in our cars to programs running on giant server farms inside big web companies—is written by humans in a higher-level programming language and then compiled into lower-level code for execution. Much of the technology for doing this translation for modern programming languages owes its beginnings to Aho and Ullman.

Beginning with their collaboration at Bell Labs in 1967 and continuing for several decades, Aho and Ullman have shaped the foundations of programming language theory and implementation, as well as algorithm design and analysis. They made broad and fundamental contributions to the field of programming language compilers through their technical contributions and influential textbooks. Their early joint work in algorithm design and analysis techniques contributed crucial approaches to the theoretical core of computer science that emerged during this period.

"The practice of computer programming and the development of increasingly advanced software systems underpin almost all of the technological transformations we have experienced in society over the last five decades," explains ACM President Gabriele Kotsis. "While countless researchers and practitioners have contributed to these technologies, the work of Aho and Ullman has been especially influential. They have helped us to understand the theoretical foundations of algorithms and to chart the course for research and practice in compilers and programming language design. Aho and Ullman have been thought leaders since the early 1970s, and their work has guided generations of programmers and researchers up to the present day."

"Aho and Ullman established bedrock ideas about algorithms, formal languages, compilers and databases, which were instrumental in the development of today's programming and software landscape," added Jeff Dean, Google Senior Fellow and SVP, Google AI. "They have also illustrated how these various disciplines are closely interconnected. Aho and Ullman introduced key technical concepts, including specific algorithms, that have been essential. In terms of computer science education, their textbooks have been the gold standard for training students, researchers, and practitioners."

A Longstanding Collaboration

Aho and Ullman both earned their PhD degrees at Princeton University before joining Bell Labs, where they worked together from 1967 to 1969. During their time at Bell Labs, their early efforts included developing efficient algorithms for analyzing and translating programming languages.

In 1969, Ullman began a career in academia, ultimately joining the faculty at Stanford University, while Aho remained at Bell Labs for 30 years before joining the faculty at Columbia University. Despite working at different institutions, Aho and Ullman continued their collaboration for several decades, during which they co-authored books and papers and introduced novel techniques for algorithms, programming languages, compilers and software systems.

Influential Textbooks

Aho and Ullman co-authored nine influential books (including first and subsequent editions). Two of their most widely celebrated books include:

The Design and Analysis of Computer Algorithms (1974)
Co-authored by Aho, Ullman, and John Hopcroft, this book is considered a classic in the field and was one of the most cited books in computer science research for more than a decade. It became the standard textbook for algorithms courses throughout the world when computer science was still an emerging field. In addition to incorporating their own research contributions to algorithms, *The Design and Analysis of Computer Algorithms* introduced the random access machine (RAM) as the basic model for analyzing the time and space complexity of computer algorithms using recurrence relations. The RAM model also codified disparate individual algorithms into general design methods. The RAM model and general algorithm design techniques introduced in this book now form an integral part of the standard computer science curriculum.

Principles of Compiler Design (1977)
Co-authored by Aho and Ullman, this definitive book on compiler technology integrated formal language theory and syntax-directed translation techniques into the compiler design process. Often called the "Dragon Book" because of its cover design, it lucidly lays out the phases in translating a high-level programming language to machine code, modularizing the entire enterprise of compiler construction. It includes algorithmic contributions that the authors made to efficient techniques for lexical analysis, syntax analysis techniques, and code generation. The current edition of this book, *Compilers: Principles, Techniques and Tools* (co-authored with Ravi Sethi and Monica Lam), was published in 2007 and remains the standard textbook on compiler design.

2020 ACM A.M. Turing Award Laureates

Alfred Vaino Aho is the Lawrence Gussman Professor Emeritus at Columbia University. He joined the Department of Computer Science at Columbia in 1995. Prior to Columbia, Aho was Vice President of Computing Sciences Research at Bell Laboratories where he worked for more than 30 years. A graduate of the University of Toronto, Aho earned his Master's and PhD degrees in Electrical Engineering/Computer Science from Princeton University.

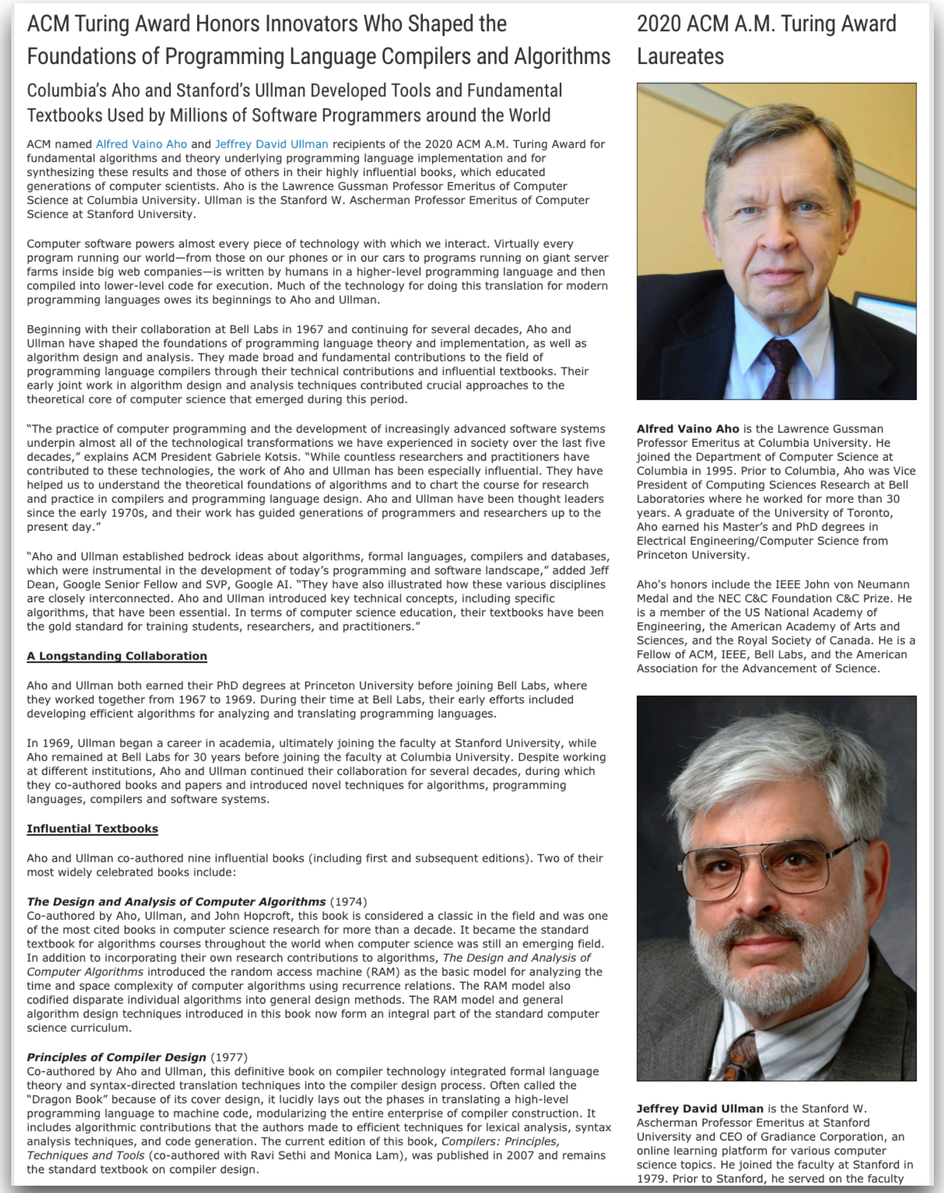
Aho's honors include the IEEE John von Neumann Medal and the NEC C&C Foundation C&C Prize. He is a member of the US National Academy of Engineering, the American Academy of Arts and Sciences, and the Royal Society of Canada. He is a Fellow of ACM, IEEE, Bell Labs, and the American Association for the Advancement of Science.

Jeffrey David Ullman is the Stanford W. Ascherman Professor Emeritus at Stanford University and CEO of Gradiance Corporation, an online learning platform for various computer science topics. He joined the faculty at Stanford in 1979. Prior to Stanford, he served on the faculty

2020 A.M. Turing Award

Turing awards by research subject (ACM)

- **Analysis of algorithms**
  - Knuth, Donald ("Don") Ervin (1974)
    - For his major contributions to the analysis of algorithms and the design of programming languages, and in particular for his contributions to the "art of computer programming" through his well-known books in a continuous series by this title.
  - ***Hopcroft, John E*** (1986)
    - With Robert E Tarjan, for fundamental achievements in the design and analysis of algorithms and data structures.
  - Tarjan, Robert (Bob) Endre (1986)
    - With John E Hopcroft, for fundamental achievements in the design and analysis of algorithms and data structures.
  - Pearl, Judea (2011)
    - For fundamental contributions to artificial intelligence through the development of a calculus for probabilistic and causal reasoning. (Bayesian networks)



ACM Turing Award Honors Innovators Who Shaped the Foundations of Programming Language Compilers and Algorithms

Columbia's Aho and Stanford's Ullman Developed Tools and Fundamental Textbooks Used by Millions of Software Programmers around the World

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

Computer software powers almost every piece of technology with which we interact. Virtually every program running our world—from those on our phones or in our cars to programs running on giant server farms inside big web companies—is written by humans in a higher-level programming language and then compiled into lower-level code for execution. Much of the technology for doing this translation for modern programming languages owes its beginnings to Aho and Ullman.

Beginning with their collaboration at Bell Labs in 1967 and continuing for several decades, Aho and Ullman have shaped the foundations of programming language theory and implementation, as well as algorithm design and analysis. They made broad and fundamental contributions to the field of programming language compilers through their technical contributions and influential textbooks. Their early joint work in algorithm design and analysis techniques contributed crucial approaches to the theoretical core of computer science that emerged during this period.

"The practice of computer programming and the development of increasingly advanced software systems underpin almost all of the technological transformations we have experienced in society over the last five decades," explains ACM President Gabriele Kotsis. "While countless researchers and practitioners have contributed to these technologies, the work of Aho and Ullman has been especially influential. They have helped us to understand the theoretical foundations of algorithms and to chart the course for research and practice in compilers and programming language design. Aho and Ullman have been thought leaders since the early 1970s, and their work has guided generations of programmers and researchers up to the present day."

"Aho and Ullman established bedrock ideas about algorithms, formal languages, compilers and databases, which were instrumental in the development of today's programming and software landscape," added Jeff Dean, Google Senior Fellow and SVP, Google AI. "They have also illustrated how these various disciplines are closely interconnected. Aho and Ullman introduced key technical concepts, including specific algorithms, that have been essential. In terms of computer science education, their textbooks have been the gold standard for training students, researchers, and practitioners."

A Longstanding Collaboration

Aho and Ullman both earned their PhD degrees at Princeton University before joining Bell Labs, where they worked together from 1967 to 1969. During their time at Bell Labs, their early efforts included developing efficient algorithms for analyzing and translating programming languages.

In 1969, Ullman began a career in academia, ultimately joining the faculty at Stanford University, while Aho remained at Bell Labs for 30 years before joining the faculty at Columbia University. Despite working at different institutions, Aho and Ullman continued their collaboration for several decades, during which they co-authored books and papers and introduced novel techniques for algorithms, programming languages, compilers and software systems.

Influential Textbooks

Aho and Ullman co-authored nine influential books (including first and subsequent editions). Two of their most widely celebrated books include:

The Design and Analysis of Computer Algorithms (1974)
Co-authored by Aho, Ullman, and John Hopcroft, this book is considered a classic in the field and was one of the most cited books in computer science research for more than a decade. It became the standard textbook for algorithms courses throughout the world when computer science was still an emerging field. In addition to incorporating their own research contributions to algorithms, *The Design and Analysis of Computer Algorithms* introduced the random access machine (RAM) as the basic model for analyzing the time and space complexity of computer algorithms using recurrence relations. The RAM model also codified disparate individual algorithms into general design methods. The RAM model and general algorithm design techniques introduced in this book now form an integral part of the standard computer science curriculum.

Principles of Compiler Design (1977)
Co-authored by Aho and Ullman, this definitive book on compiler technology integrated formal language theory and syntax-directed translation techniques into the compiler design process. Often called the "Dragon Book" because of its cover design, it lucidly lays out the phases in translating a high-level programming language to machine code, modularizing the entire enterprise of compiler construction. It includes algorithmic contributions that the authors made to efficient techniques for lexical analysis, syntax analysis techniques, and code generation. The current edition of this book, *Compilers: Principles, Techniques and Tools* (co-authored with Ravi Sethi and Monica Lam), was published in 2007 and remains the standard textbook on compiler design.

2020 ACM A.M. Turing Award Laureates

Alfred Vaino Aho is the Lawrence Gussman Professor Emeritus at Columbia University. He joined the Department of Computer Science at Columbia in 1995. Prior to Columbia, Aho was Vice President of Computing Sciences Research at Bell Laboratories where he worked for more than 30 years. A graduate of the University of Toronto, Aho earned his Master's and PhD degrees in Electrical Engineering/Computer Science from Princeton University.

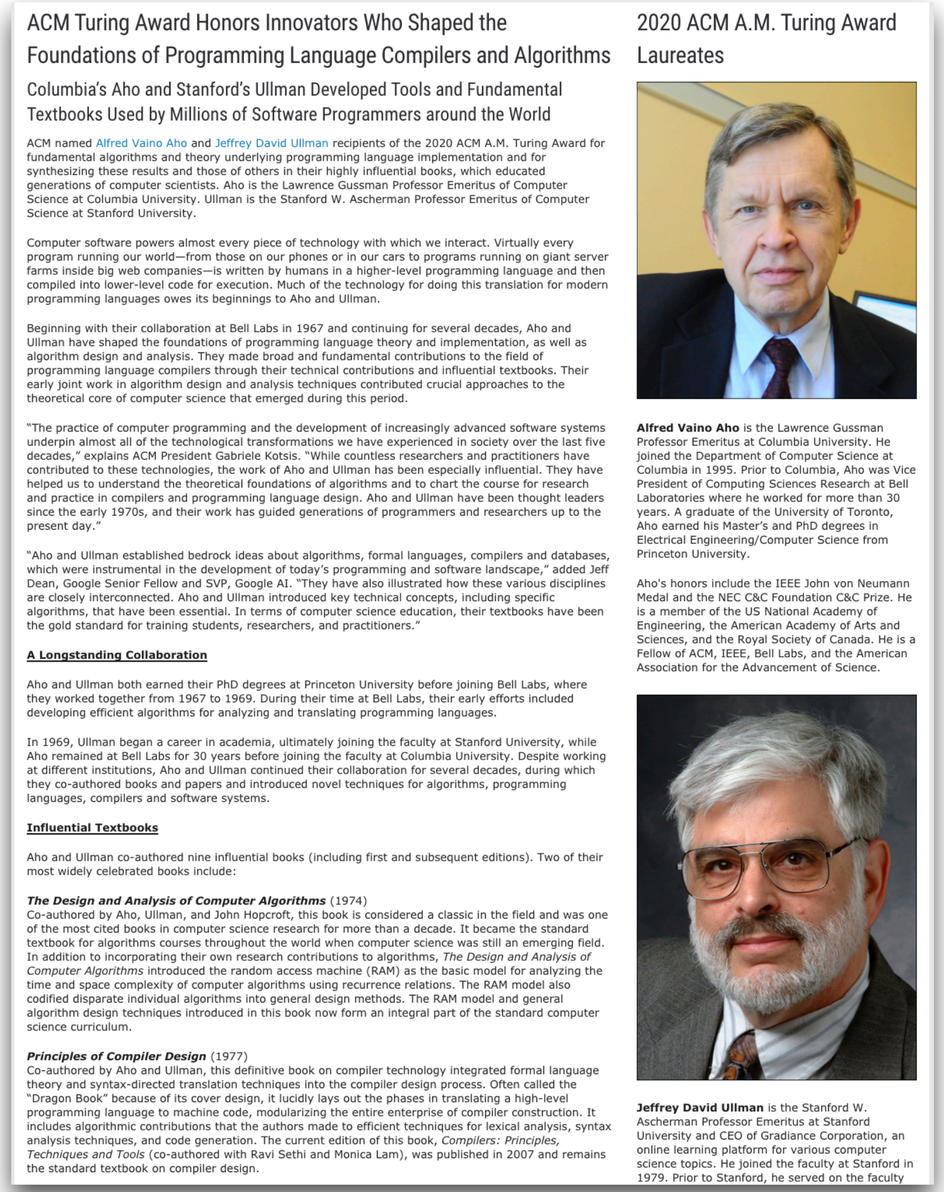
Aho's honors include the IEEE John von Neumann Medal and the NEC C&C Foundation C&C Prize. He is a member of the US National Academy of Engineering, the American Academy of Arts and Sciences, and the Royal Society of Canada. He is a Fellow of ACM, IEEE, Bell Labs, and the American Association for the Advancement of Science.

Jeffrey David Ullman is the Stanford W. Ascherman Professor Emeritus at Stanford University and CEO of Gradiance Corporation, an online learning platform for various computer science topics. He joined the faculty at Stanford in 1979. Prior to Stanford, he served on the faculty

2020 A.M. Turing Award

Turing awards by research subject (ACM)

- **Programming languages**
  - Iverson, Kenneth E. ("Ken") (1979)
    - For his pioneering effort in programming languages and mathematical notation resulting in what the computing field now knows as APL, for his contributions to the implementation of interactive systems, to educational uses of APL, and to programming language theory and practice.
  - Milner, Arthur John Robin Gorell ("Robin") (1991)
    - For three distinct and complete achievements: (i) LCF, the mechanization of Scott's Logic of Computable Functions, probably the first theoretically based yet practical tool for machine assisted proof construction; (ii) ML, the first language to include polymorphic type inference together with a type-safe exception-handling mechanism; (iii) CCS, a general theory of concurrency. In addition, he formulated and strongly advanced full abstraction, the study of the relationship between operational and denotational semantics.
  - Liskov, Barbara (2008)
    - For contributions to practical and theoretical foundations of programming language and system design, especially related to data abstraction, fault tolerance, and distributed computing.
  - Hoare, C. Antony ("Tony") R. (1980)
    - For his fundamental contributions to the definition and design of programming languages.
  - Kay, Alan (2003)
    - For pioneering many of the ideas at the root of contemporary object-oriented programming languages, leading the team that developed Smalltalk, and for fundamental contributions to personal computing.



ACM Turing Award Honors Innovators Who Shaped the Foundations of Programming Language Compilers and Algorithms

Columbia's Aho and Stanford's Ullman Developed Tools and Fundamental Textbooks Used by Millions of Software Programmers around the World

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

Computer software powers almost every piece of technology with which we interact. Virtually every program running our world—from those on our phones or in our cars to programs running on giant server farms inside big web companies—is written by humans in a higher-level programming language and then compiled into lower-level code for execution. Much of the technology for doing this translation for modern programming languages owes its beginnings to Aho and Ullman.

Beginning with their collaboration at Bell Labs in 1967 and continuing for several decades, Aho and Ullman have shaped the foundations of programming language theory and implementation, as well as algorithm design and analysis. They made broad and fundamental contributions to the field of programming language compilers through their technical contributions and influential textbooks. Their early joint work in algorithm design and analysis techniques contributed crucial approaches to the theoretical core of computer science that emerged during this period.

"The practice of computer programming and the development of increasingly advanced software systems underpin almost all of the technological transformations we have experienced in society over the last five decades," explains ACM President Gabriele Kotsis. "While countless researchers and practitioners have contributed to these technologies, the work of Aho and Ullman has been especially influential. They have helped us to understand the theoretical foundations of algorithms and to chart the course for research and practice in compilers and programming language design. Aho and Ullman have been thought leaders since the early 1970s, and their work has guided generations of programmers and researchers up to the present day."

"Aho and Ullman established bedrock ideas about algorithms, formal languages, compilers and databases, which were instrumental in the development of today's programming and software landscape," added Jeff Dean, Google Senior Fellow and SVP, Google AI. "They have also illustrated how these various disciplines are closely interconnected. Aho and Ullman introduced key technical concepts, including specific algorithms, that have been essential. In terms of computer science education, their textbooks have been the gold standard for training students, researchers, and practitioners."

A Longstanding Collaboration

Aho and Ullman both earned their PhD degrees at Princeton University before joining Bell Labs, where they worked together from 1967 to 1969. During their time at Bell Labs, their early efforts included developing efficient algorithms for analyzing and translating programming languages.

In 1969, Ullman began a career in academia, ultimately joining the faculty at Stanford University, while Aho remained at Bell Labs for 30 years before joining the faculty at Columbia University. Despite working at different institutions, Aho and Ullman continued their collaboration for several decades, during which they co-authored books and papers and introduced novel techniques for algorithms, programming languages, compilers and software systems.

Influential Textbooks

Aho and Ullman co-authored nine influential books (including first and subsequent editions). Two of their most widely celebrated books include:

The Design and Analysis of Computer Algorithms (1974)
Co-authored by Aho, Ullman, and John Hopcroft, this book is considered a classic in the field and was one of the most cited books in computer science research for more than a decade. It became the standard textbook for algorithms courses throughout the world when computer science was still an emerging field. In addition to incorporating their own research contributions to algorithms, *The Design and Analysis of Computer Algorithms* introduced the random access machine (RAM) as the basic model for analyzing the time and space complexity of computer algorithms using recurrence relations. The RAM model also codified disparate individual algorithms into general design methods. The RAM model and general algorithm design techniques introduced in this book now form an integral part of the standard computer science curriculum.

Principles of Compiler Design (1977)
Co-authored by Aho and Ullman, this definitive book on compiler technology integrated formal language theory and syntax-directed translation techniques into the compiler design process. Often called the "Dragon Book" because of its cover design, it lucidly lays out the phases in translating a high-level programming language to machine code, modularizing the entire enterprise of compiler construction. It includes algorithmic contributions that the authors made to efficient techniques for lexical analysis, syntax analysis techniques, and code generation. The current edition of this book, *Compilers: Principles, Techniques and Tools* (co-authored with Ravi Sethi and Monica Lam), was published in 2007 and remains the standard textbook on compiler design.

2020 ACM A.M. Turing Award Laureates

Alfred Vaino Aho is the Lawrence Gussman Professor Emeritus at Columbia University. He joined the Department of Computer Science at Columbia in 1995. Prior to Columbia, Aho was Vice President of Computing Sciences Research at Bell Laboratories where he worked for more than 30 years. A graduate of the University of Toronto, Aho earned his Master's and PhD degrees in Electrical Engineering/Computer Science from Princeton University.

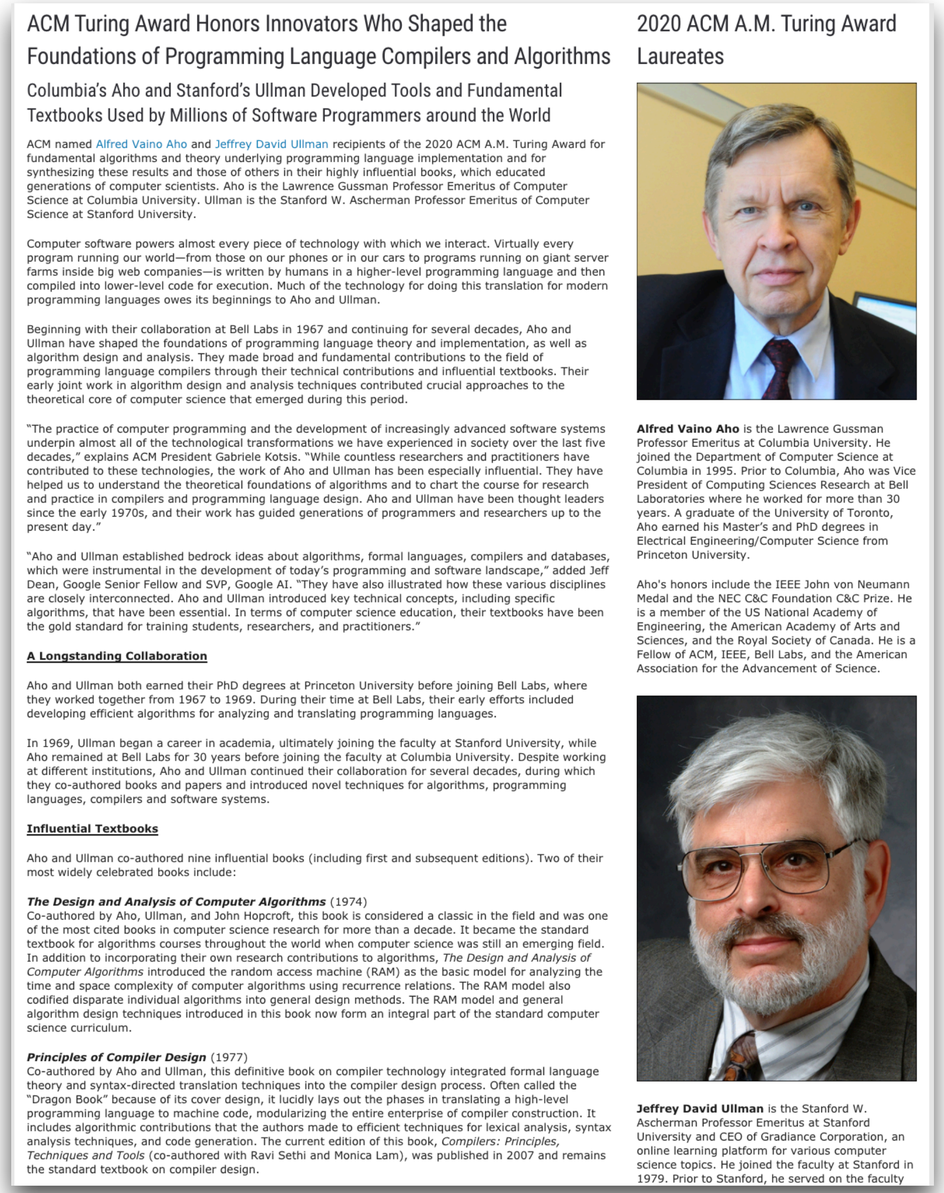
Aho's honors include the IEEE John von Neumann Medal and the NEC C&C Foundation C&C Prize. He is a member of the US National Academy of Engineering, the American Academy of Arts and Sciences, and the Royal Society of Canada. He is a Fellow of ACM, IEEE, Bell Labs, and the American Association for the Advancement of Science.

Jeffrey David Ullman is the Stanford W. Ascherman Professor Emeritus at Stanford University and CEO of Gradiance Corporation, an online learning platform for various computer science topics. He joined the faculty at Stanford in 1979. Prior to Stanford, he served on the faculty

2020 A.M. Turing Award

Turing awards by research subject (ACM)

- **Compilers**
  - Perlis, Alan J (1966)
    - For his influence in the area of advanced programming techniques and compiler construction. (He was part of the team that developed ALGOL.)
  - Cocke, John (1987)
    - For significant contributions in the design and theory of compilers, the architecture of large systems and the development of reduced instruction set computers (RISC); for discovering and systematizing many fundamental transformations now used in optimizing compilers including reduction of operator strength, elimination of common subexpressions, register allocation, constant propagation, and dead code elimination.
  - Allen, Frances ("Fran") Elizabeth (2006)
    - For pioneering contributions to the theory and practice of optimizing compiler techniques that laid the foundation for modern optimizing compilers and automatic parallel execution.
  - **Aho, Alfred Vaino (2020)**
  - **Ullman, Jeffrey David (2020)**



ACM Turing Award Honors Innovators Who Shaped the Foundations of Programming Language Compilers and Algorithms

Columbia's Aho and Stanford's Ullman Developed Tools and Fundamental Textbooks Used by Millions of Software Programmers around the World

ACM named Alfred Vaino Aho and Jeffrey David Ullman recipients of the 2020 ACM A.M. Turing Award for fundamental algorithms and theory underlying programming language implementation and for synthesizing these results and those of others in their highly influential books, which educated generations of computer scientists. Aho is the Lawrence Gussman Professor Emeritus of Computer Science at Columbia University. Ullman is the Stanford W. Ascherman Professor Emeritus of Computer Science at Stanford University.

Computer software powers almost every piece of technology with which we interact. Virtually every program running our world—from those on our phones or in our cars to programs running on giant server farms inside big web companies—is written by humans in a higher-level programming language and then compiled into lower-level code for execution. Much of the technology for doing this translation for modern programming languages owes its beginnings to Aho and Ullman.

Beginning with their collaboration at Bell Labs in 1967 and continuing for several decades, Aho and Ullman have shaped the foundations of programming language theory and implementation, as well as algorithm design and analysis. They made broad and fundamental contributions to the field of programming language compilers through their technical contributions and influential textbooks. Their early joint work in algorithm design and analysis techniques contributed crucial approaches to the theoretical core of computer science that emerged during this period.

"The practice of computer programming and the development of increasingly advanced software systems underpin almost all of the technological transformations we have experienced in society over the last five decades," explains ACM President Gabriele Kotsis. "While countless researchers and practitioners have contributed to these technologies, the work of Aho and Ullman has been especially influential. They have helped us to understand the theoretical foundations of algorithms and to chart the course for research and practice in compilers and programming language design. Aho and Ullman have been thought leaders since the early 1970s, and their work has guided generations of programmers and researchers up to the present day."

"Aho and Ullman established bedrock ideas about algorithms, formal languages, compilers and databases, which were instrumental in the development of today's programming and software landscape," added Jeff Dean, Google Senior Fellow and SVP, Google AI. "They have also illustrated how these various disciplines are closely interconnected. Aho and Ullman introduced key technical concepts, including specific algorithms, that have been essential. In terms of computer science education, their textbooks have been the gold standard for training students, researchers, and practitioners."

A Longstanding Collaboration

Aho and Ullman both earned their PhD degrees at Princeton University before joining Bell Labs, where they worked together from 1967 to 1969. During their time at Bell Labs, their early efforts included developing efficient algorithms for analyzing and translating programming languages.

In 1969, Ullman began a career in academia, ultimately joining the faculty at Stanford University, while Aho remained at Bell Labs for 30 years before joining the faculty at Columbia University. Despite working at different institutions, Aho and Ullman continued their collaboration for several decades, during which they co-authored books and papers and introduced novel techniques for algorithms, programming languages, compilers and software systems.

Influential Textbooks

Aho and Ullman co-authored nine influential books (including first and subsequent editions). Two of their most widely celebrated books include:

The Design and Analysis of Computer Algorithms (1974)
Co-authored by Aho, Ullman, and John Hopcroft, this book is considered a classic in the field and was one of the most cited books in computer science research for more than a decade. It became the standard textbook for algorithms courses throughout the world when computer science was still an emerging field. In addition to incorporating their own research contributions to algorithms, *The Design and Analysis of Computer Algorithms* introduced the random access machine (RAM) as the basic model for analyzing the time and space complexity of computer algorithms using recurrence relations. The RAM model also codified disparate individual algorithms into general design methods. The RAM model and general algorithm design techniques introduced in this book now form an integral part of the standard computer science curriculum.

Principles of Compiler Design (1977)
Co-authored by Aho and Ullman, this definitive book on compiler technology integrated formal language theory and syntax-directed translation techniques into the compiler design process. Often called the "Dragon Book" because of its cover design, it lucidly lays out the phases in translating a high-level programming language to machine code, modularizing the entire enterprise of compiler construction. It includes algorithmic contributions that the authors made to efficient techniques for lexical analysis, syntax analysis techniques, and code generation. The current edition of this book, *Compilers: Principles, Techniques and Tools* (co-authored with Ravi Sethi and Monica Lam), was published in 2007 and remains the standard textbook on compiler design.

2020 ACM A.M. Turing Award Laureates

Alfred Vaino Aho is the Lawrence Gussman Professor Emeritus at Columbia University. He joined the Department of Computer Science at Columbia in 1995. Prior to Columbia, Aho was Vice President of Computing Sciences Research at Bell Laboratories where he worked for more than 30 years. A graduate of the University of Toronto, Aho earned his Master's and PhD degrees in Electrical Engineering/Computer Science from Princeton University.

Aho's honors include the IEEE John von Neumann Medal and the NEC C&C Foundation C&C Prize. He is a member of the US National Academy of Engineering, the American Academy of Arts and Sciences, and the Royal Society of Canada. He is a Fellow of ACM, IEEE, Bell Labs, and the American Association for the Advancement of Science.

Jeffrey David Ullman is the Stanford W. Ascherman Professor Emeritus at Stanford University and CEO of Gradiance Corporation, an online learning platform for various computer science topics. He joined the faculty at Stanford in 1979. Prior to Stanford, he served on the faculty

The boys

Alfred Vaino Aho

Jeffrey David Ullman

The boys

Alfred Vaino Aho

- Alfred V. Aho is Lawrence Gussman **Professor Emeritus of Computer Science at Columbia University**. He joined the Department of Computer Science at Columbia in 1995.
- Professor Aho has a **B.A.Sc in Engineering Physics from the University of Toronto** and a **Ph.D. in Electrical Engineering/Computer Science from Princeton** University.
- Professor Aho won the **Great Teacher Award** for 2003 **from the Society of Columbia Graduates**. In 2014 he was again recognized for teaching excellence by winning the **Distinguished Faculty Teaching Award from the Columbia Engineering Alumni Association**.
- Professor Aho has received the ACM A.M. Turing Award and the **IEEE John von Neumann Medal**. He is a Member of the U.S. National Academy of Engineering and of the American Academy of Arts and Sciences. He is a Fellow of the Royal Society of Canada. He shared the 2017 C&C prize with John Hopcroft and Jeff Ullman. He has received honorary doctorates from the Universities of Helsinki, Toronto and Waterloo, and is a Fellow of the American Association for the Advancement of Science, ACM, Bell Labs, and IEEE.
- **Professor Aho is the "A" in AWK**, a widely used pattern-matching language; *"W" is Peter Weinberger and "K" is Brian Kernighan*. (Think of AWK as the predecessor of perl.) He also wrote the initial versions of the string pattern-matching utilities egrep and fgrep that are a part of UNIX; fgrep was the first widely used implementation of what is now called the Aho-Corasick algorithm.
- Prior to his position at Columbia, Professor Aho was **Vice President of the Computing Sciences Research Center at Bell Labs**, the lab that invented UNIX, C and C++. He was previously a member of technical staff, a department head, and the director of this center. Professor Aho also served as General Manager of the Information Sciences and Technologies Research Laboratory at Bellcore (now Telcordia).

Excerpt from `http://www.cs.columbia.edu/~aho/bio.html`

The boys

- Born: 22 November 1942
- Education: **B. S. in Engineering Mathematics, Columbia** Univ., 1963; **Ph. D. in Electrical Engineering, Princeton** Univ., 1966.
- Membership in Societies: NAE, NAS, AAAS, ACM, EATCS, SIGACT, SIGMOD, TBPi, SigmaXi.
- Employment History: **Bell Laboratories**, Murray Hill, NJ, 1966–1969. Member of Technical Staff. **Princeton** Univ., Princeton, NJ., Associate Professor, 1969--1974, Professor, 1974–1979, Stanford Univ., Stanford, CA, Professor, 1979–2002. **S. W. Ascherman Prof. of Engineering**, 1994--2002 (emeritus 2003—present), Chair of Department of Computer Science, 1990–1994, **Gradiance Corp.**, Stanford CA, CEO, 2003—present.
- Honors: **Honorary doctorate**, Free University of Brussels, 1975; Einstein Fellowship, Israeli Academy of Sciences, 1984; Guggenheim Fellowship, 1988–89; National Academy of Engineering, 1989; Honorary doctorate, University of Paris-Dauphine, 1992; **Fellow** of Association for Computing Machinery, 1994, SIGMOD Contributions Award, 1996, Best paper award, SIGMOD, 1996, Karl V. Karlstrom outstanding educator award, ACM, 1998, Knuth Prize, 2000, E.F. Codd Innovation Award, SIGMOD, 2006, **Test-of-Time Award**, SIGMOD, 2006, **von Neumann Medal**, IEEE, 2010, American Academy of Arts and Sciences, 2012, Honorary doctorate, Ben Gurion University, 2016, NEC C&C Foundation Prize, 2017, National Academy of Sciences, 2020, Test-of-Time Award, EDBT, 2020, A.M. Turing Award, 2020.

  Excerpt `from http://infolab.stanford.edu/~ullman/pub/opb.txt`

Jeffrey David Ullman

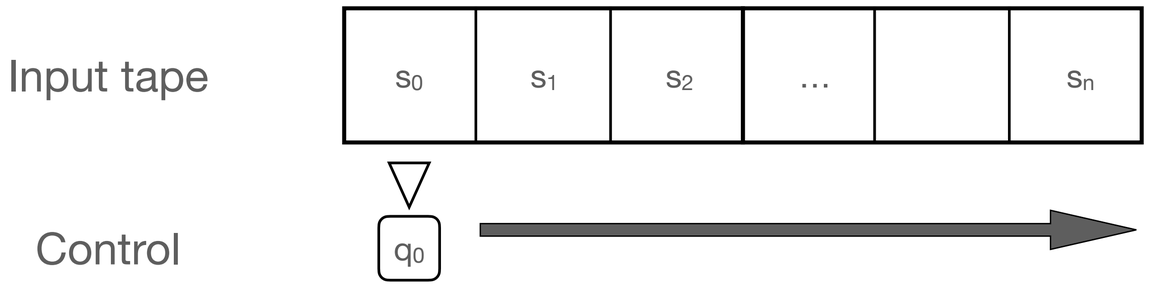
Foundations of programming language compilers

Background

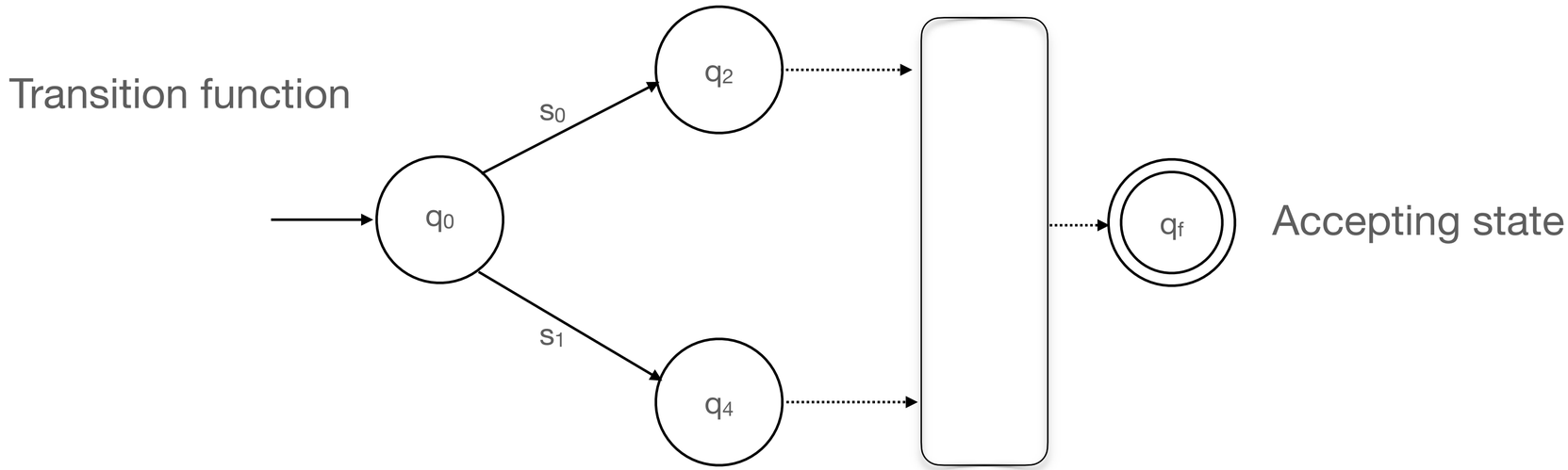
- Automata theory
- Languages theory
- Compiler principles

Foundations of programming language compilers

Automata theory > Deterministic Finite Automata



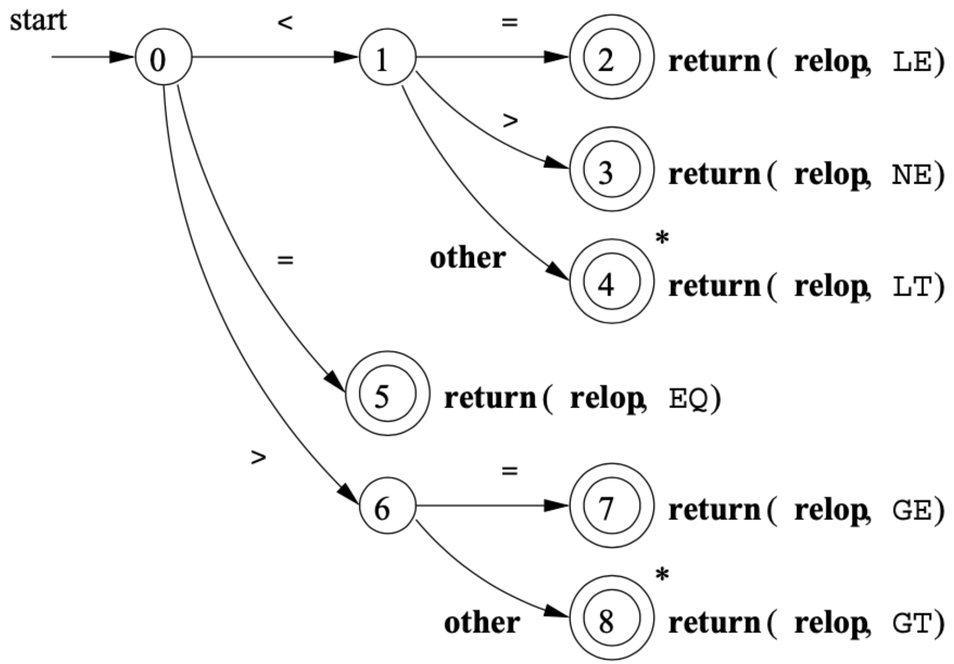
$$\Sigma = \{s_0, s_1, \ldots, s_n\}, L \subseteq \Sigma^*$$



Foundations of programming language compilers

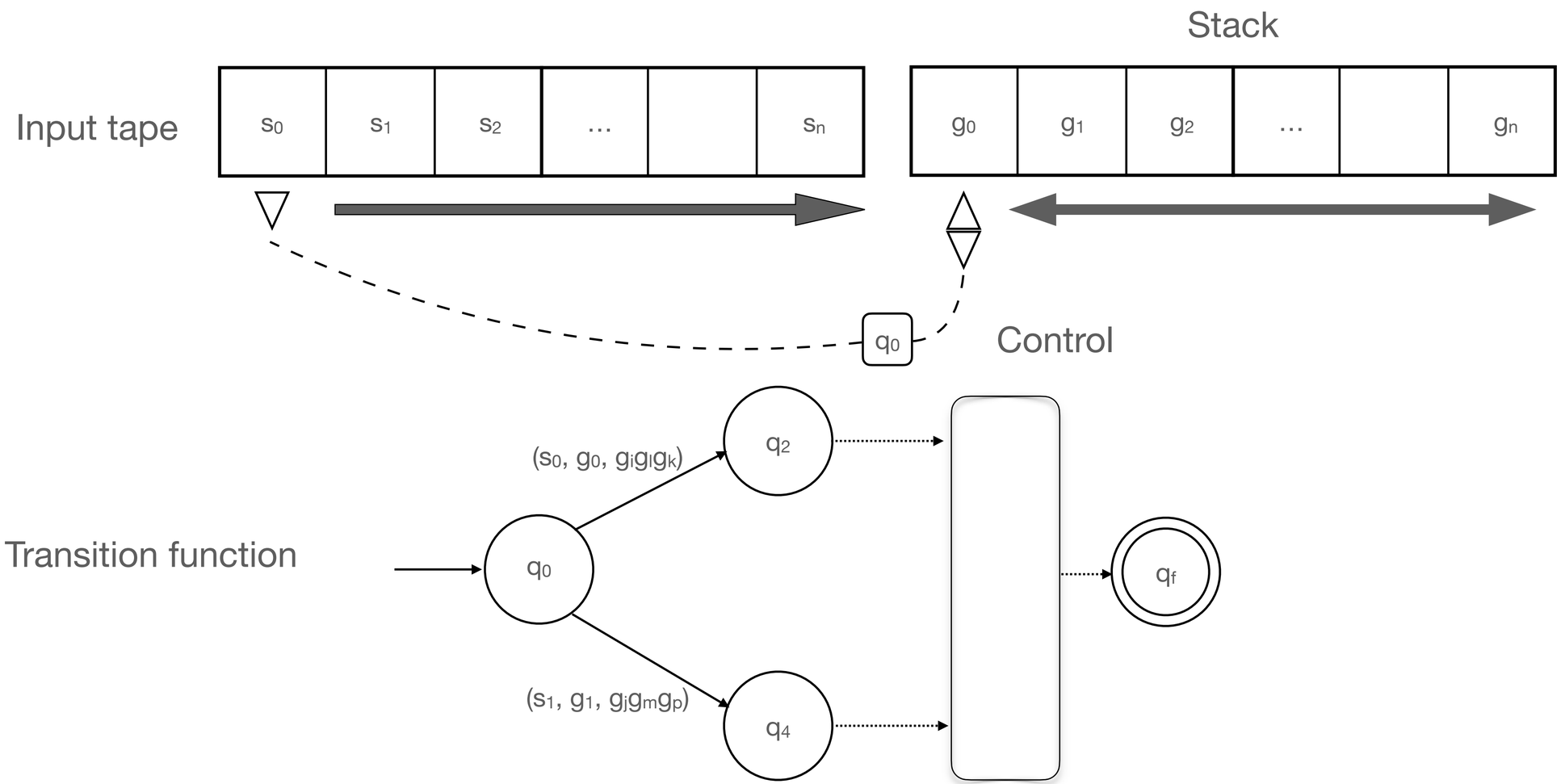
Automata theory > Deterministic Finite Automata > Exemple

AFD for binary operations



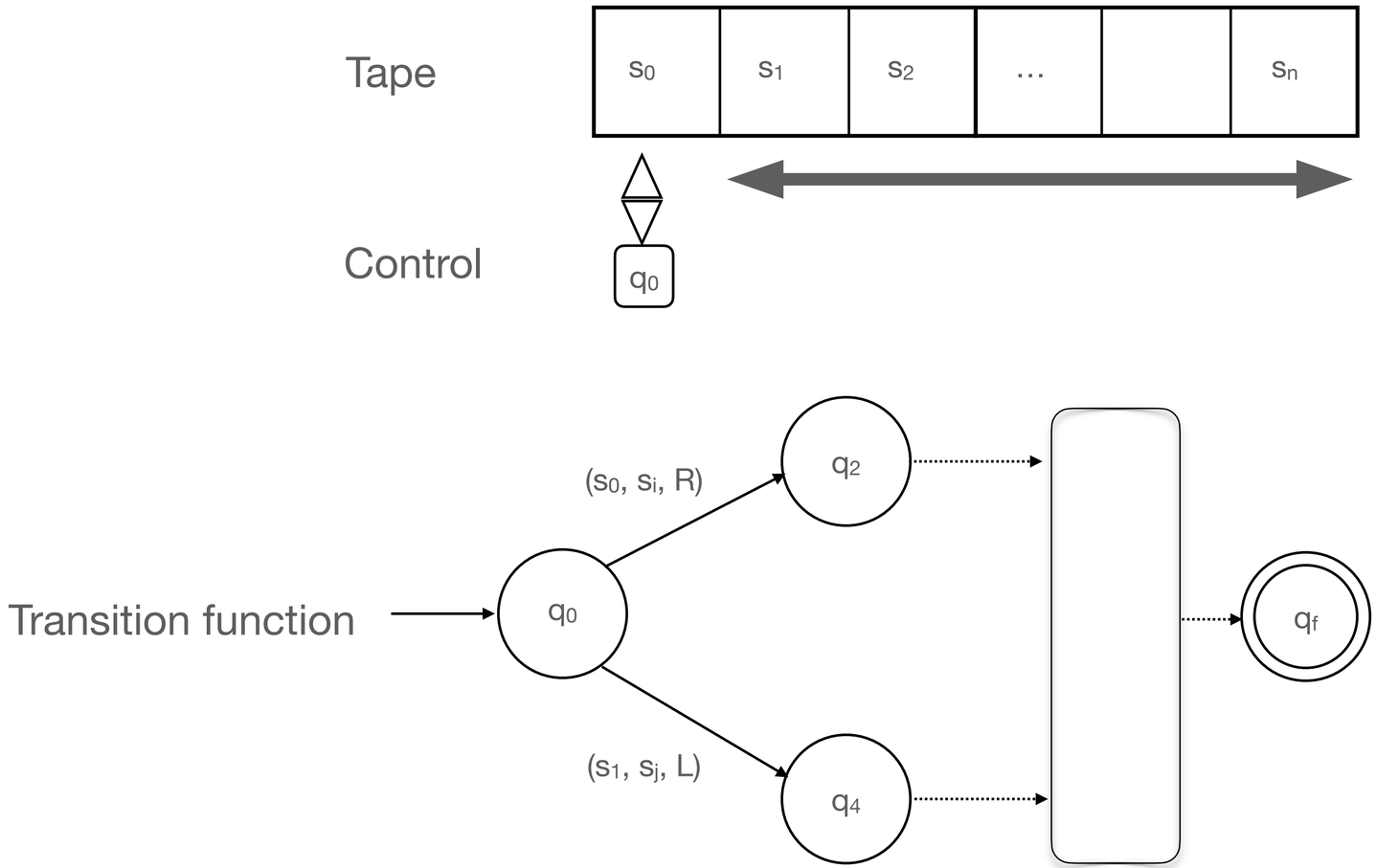
Foundations of programming language compilers

Automata theory > Pushdown Automata



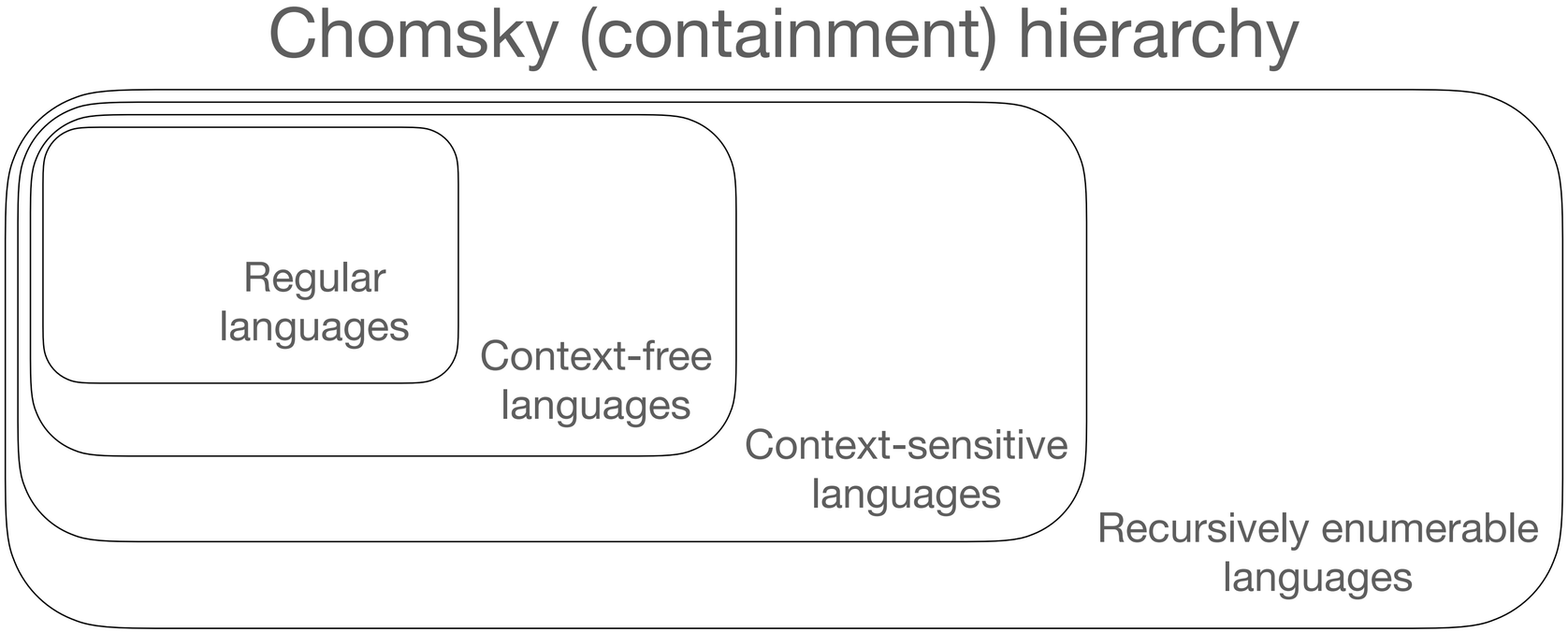
Foundations of programming language compilers

Automata theory > Turing machine



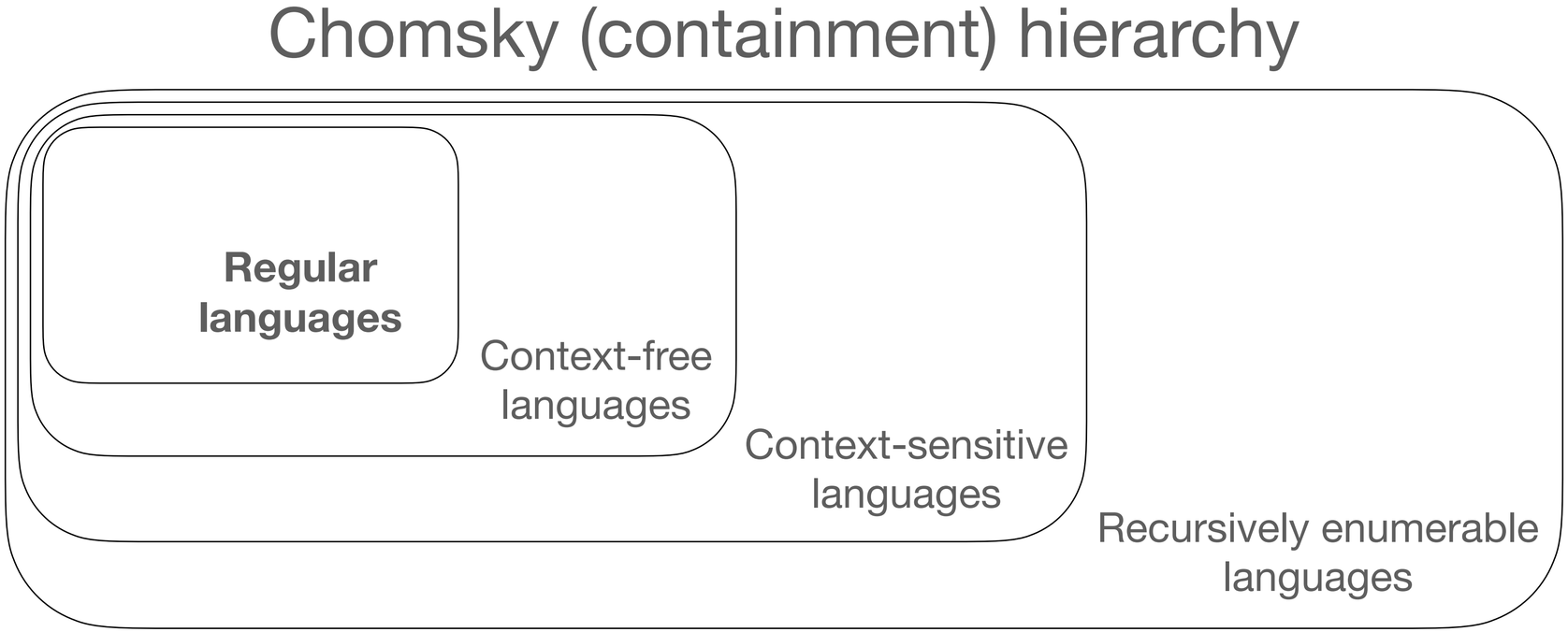
Foundations of programming language compilers

Language theory



Foundations of programming language compilers

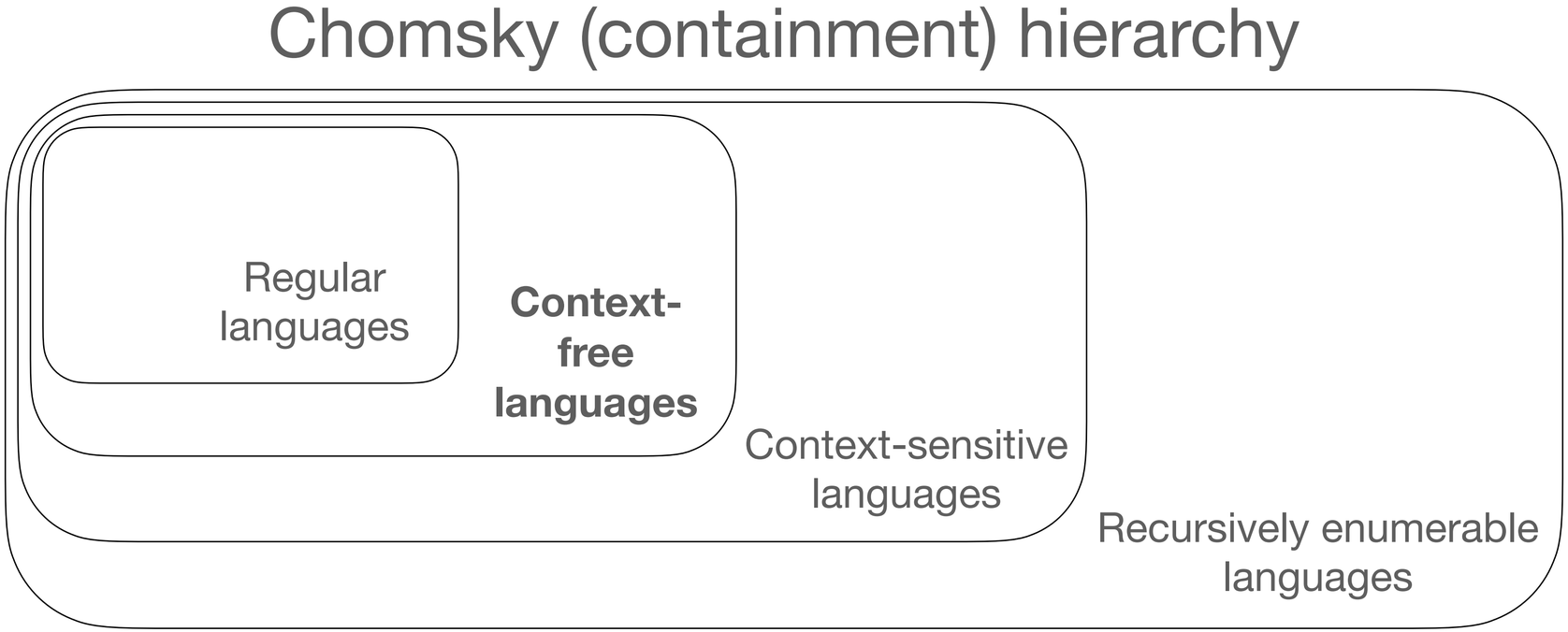
Language theory



- Generated by linear grammars
- Accepted by DFA

Foundations of programming language compilers

Language theory



- Generated by context-free grammars
- Accepted by pushdown automata

Foundations of programming language compilers

Language theory > Context-free language grammar

CFG for arithmetic expressions

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

$$E \Rightarrow E + E \Rightarrow id + E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$$

$$E \Rightarrow E * E \Rightarrow E + E * E \Rightarrow id + E * E \Rightarrow id + id * E \Rightarrow id + id * id$$

Foundations of programming language compilers

Language theory > Context-free language grammar

CFG for arithmetic expressions

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

CFG for arithmetic expressions with precedence

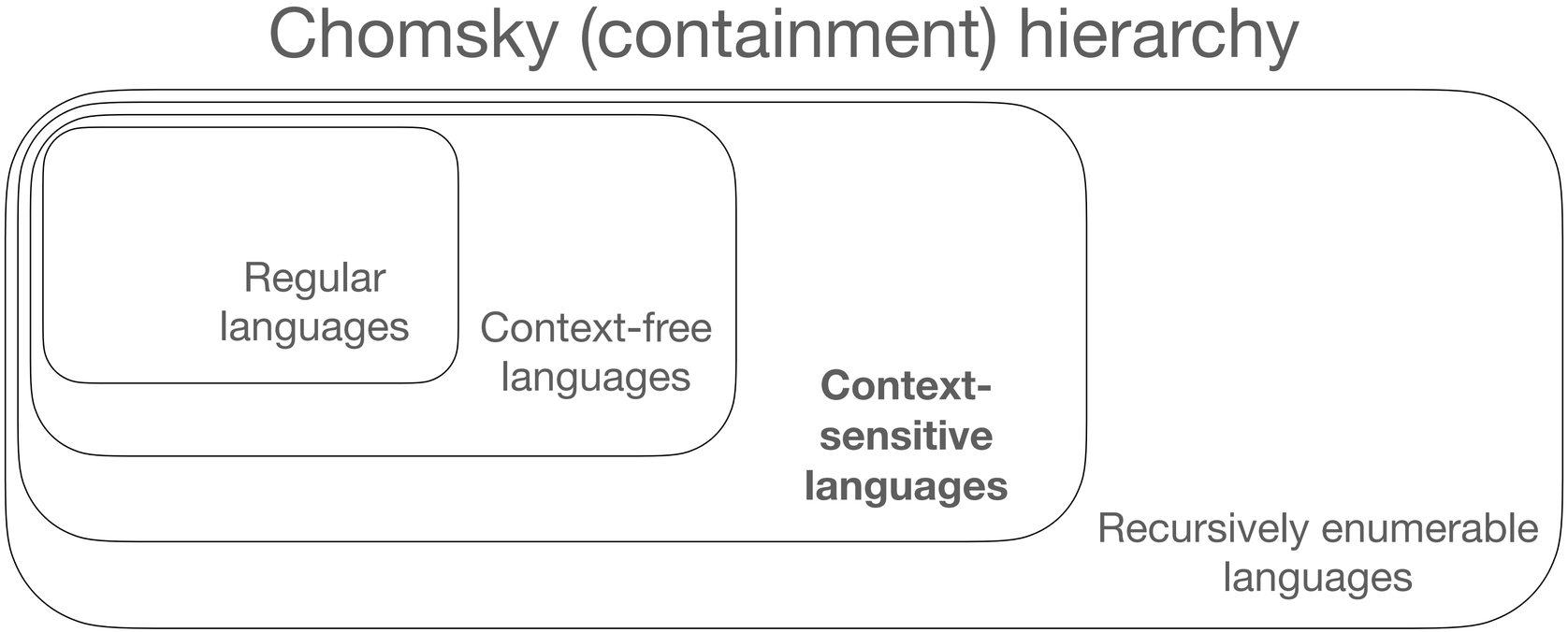
$$\begin{aligned} E &\rightarrow E + T \mid T \\ T &\rightarrow T * F \mid F \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

CFG for arithmetic expressions without left-recursion

$$\begin{aligned} E &\rightarrow T\ E' \\ E' &\rightarrow +\ T\ E' \mid \epsilon \\ T &\rightarrow F\ T' \\ T' &\rightarrow *\ F\ T' \mid \epsilon \\ F &\rightarrow (E) \mid \mathbf{id} \end{aligned}$$

Foundations of programming language compilers

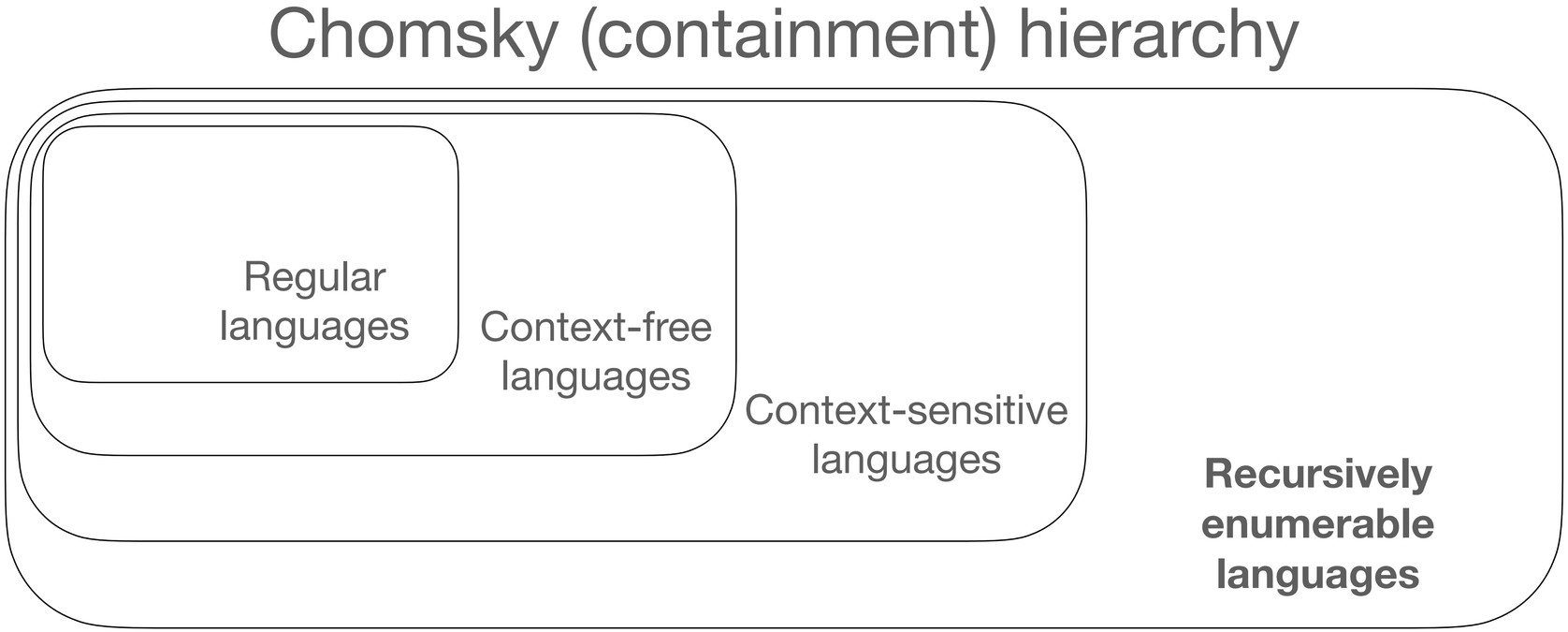
Language theory



- Generated by context-sensitive grammars
- Accepted by linear-bounded automata

Foundations of programming language compilers

Language theory



- Generated by unrestricted grammars
- Accepted by Turing machines

Foundations of programming language compilers

Language theory > CFG and PA are two sides of the same coin

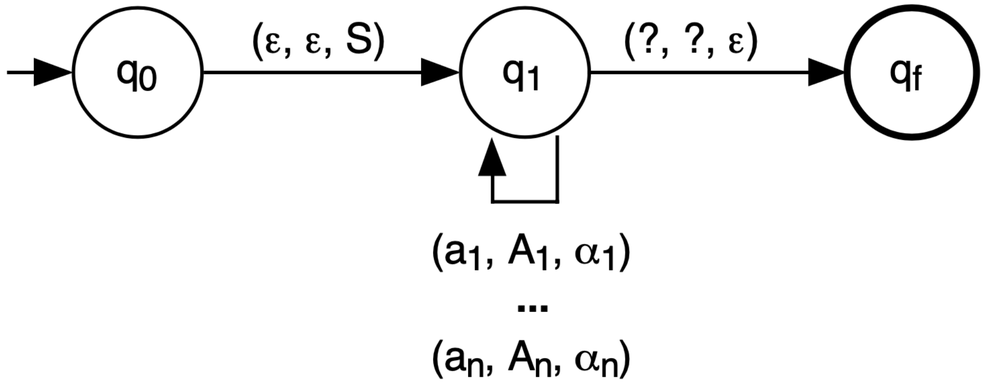
Context-free languages

Generated by context-free grammars

Accepted by Pushdown Automata

$A \rightarrow \alpha$

$A \rightarrow aA_1A_2 \ldots A_n$, Greibach normal form



Foundations of programming language compilers

Language theory > $L = \{a^n b^n c^n \mid n \geq 1\}$ is not a CFL

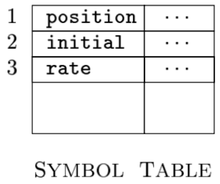
- This result is by the application of the *pumping lemma* for CFL.
- If L was a CFL, for a certain number n, it would be possible to decompose all strings $w, |w| \geq n,$ as $w = uvxyz$ such that u or v is not empty and $uv^i xy^i z$, for all $i \geq 0$.
- The point is that there is no such decomposition that keeps the number of a, b, and c balanced when a string w is “pumped”.
- An if-then-else statement can not be specified by a CFG.

Foundations of programming language compilers

Compiler principles

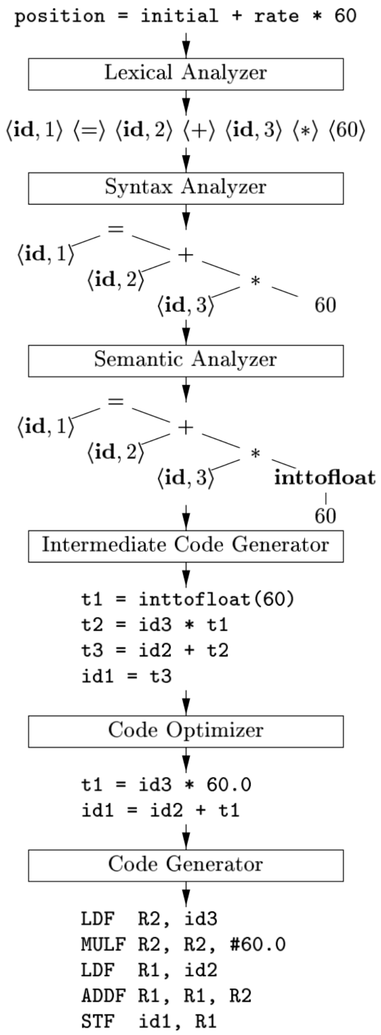
Regular expressions

Syntax-directed translation



1	position	···
2	initial	···
3	rate	···

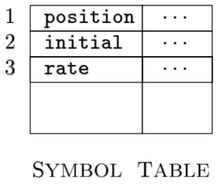
SYMBOL TABLE



Foundations of programming language compilers

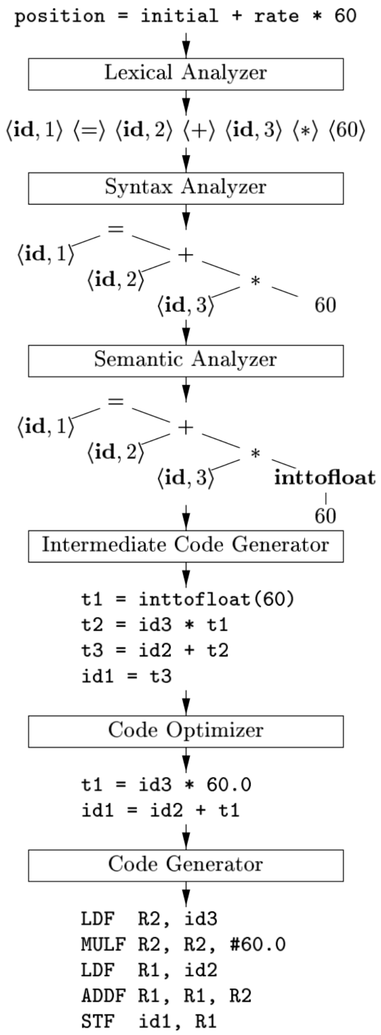
Compiler principles

Syntax-directed translation



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Context-free grammar

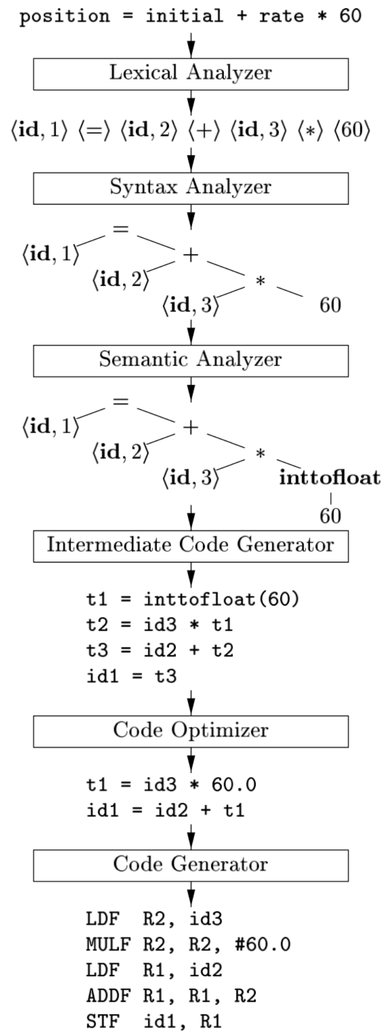
Foundations of programming language compilers

Compiler principles

Syntax-directed translation

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE

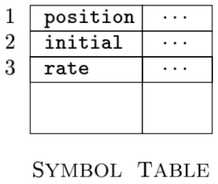


Attribute grammar

Foundations of programming language compilers

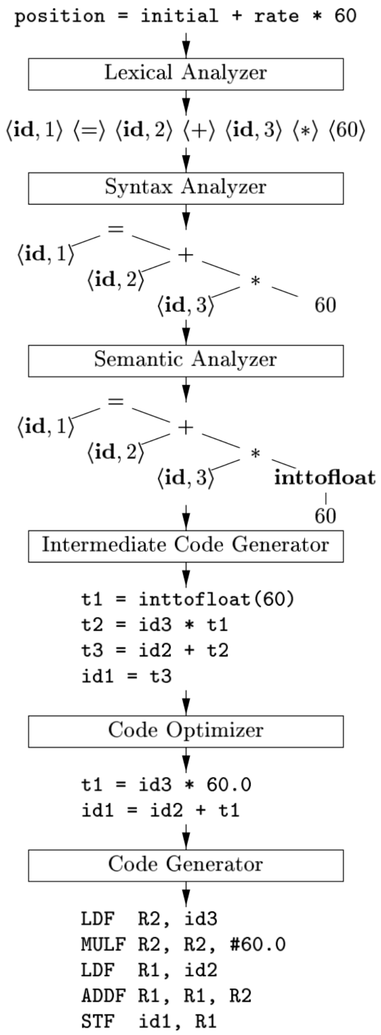
Compiler principles

Syntax-directed translation



1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Attribute grammar

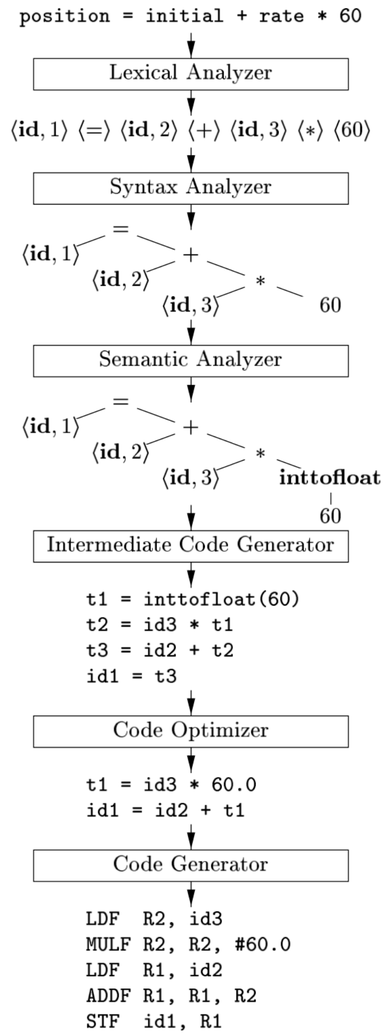
Foundations of programming language compilers

Compiler principles

Syntax-directed translation

1	position	···
2	initial	···
3	rate	···

SYMBOL TABLE



Attribute grammar

PhD work

Alfred Aho

- Indexed Grammars - an extension of context-free grammars
- Princeton, 1967
- Supervisor: John Hopcroft (Turing laureate, 1986)
- Aho, A. V. (1968). Indexed Grammars - an extension of context-free grammars. Journal of the ACM, 15(4), pp. 647-671. https://doi.org/10.1145/321479.321488
- Aho, A. V. (1969). Nested Stack Automata. Journal of the ACM, 16(3), pp. 383–406. https://doi.org/10.1145/321526.321529

Indexed Grammars—An Extension of Context-Free Grammars

ALFRED V. AHO

Bell Telephone Laboratories, Inc., Murray Hill,

ABSTRACT. A new type of grammar for generati
mar, is presented. An indexed grammar is an e
class of languages generated by indexed gramm
sults similar to those for context-free languages
grammars properly includes all context-free lan
context-sensitive languages. Several subclasses
classes of languages.

KEY WORDS AND PHRASES: formal grammar,
theory, phrase-structure grammar, phrase-stru
text-free grammar, context-sensitive grammar,

CR CATEGORIES: 5.22, 5.23

1. *Introduction*

A language, whether a natural language su
such as ALGOL, in an abstract sense can
One criterion for the syntactic specificati
finite representation is required for an infi
different approaches toward such a specific
tive device, called a grammar, is used to d
guage. Another approach is to specify a d
formed sentences. In this approach the lang
by the device or algorithm. A third possib
guage would be to specify a set of properti
set of words obeying these properties.

In this paper a new type of grammar,
The language generated by an indexed gra
is shown that the class of indexed langu
languages and yet is a proper subset of t
Moreover, indexed languages resemble con
closure properties and decidability results f

Recently, there has been a great deal of
languages larger than the class of context
are a recent example of a grammatical def

A summary of this paper was presented at the
ing and Automata Theory, October 1967.
[1] The words "sentence," "word," and "program"
ment in a language.

Journal of the Association for Computing Machinery, Vol. 15,

Copyright © 1968, Association for Computing Machinery, Inc.

Nested Stack Automata

ALFRED V. AHO

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. A new type of automaton, called a nested stack automaton, is introduced. The nested stack automaton provides a mathematical model for a restricted form of embedded list structure. Special cases of nested stack automata include many of the important machine models studied in automata theory, such as pushdown automata and stack automata.

The fundamental properties of languages accepted by nested stack automata are investigated. It is shown that the class of languages accepted by one-way nondeterministic nested stack automata is the same as the class of indexed languages.

KEY WORDS AND PHRASES: automata, formal languages, indexed languages, pushdown automata, stack automata, acceptors, balloon automata, closure properties, decidability results, list storage

CR CATEGORIES: 5.21, 5.22

1. *Introduction*

A new kind of memory structure, called a nested stack, is introduced. A nested stack provides a mathematical model for a restricted form of embedded list structure. The computational power of this structure is investigated through the vehicle of an abstract machine, called a nested stack automaton, which consists of a finite-state machine with a nested stack as the main auxiliary memory.

Special cases of a nested stack automaton include the pushdown automaton and the recently introduced stack automaton [7, 8]. It is shown that nested stack automata have less computational capability than Turing machines and are, in fact, recursive. The family of nested stack automata is equivalent to one of the largest known closed classes of balloon automata which define recursive sets.

A language can be finitely specified as the set of finite length strings of symbols either generated by some generative device, as a grammar, or accepted by some recognitive device, as an abstract machine. It is shown that the class of indexed languages [1] is exactly that class of sets accepted by one-way nondeterministic nested stack automata. It is also shown that a characterization for one-way nondeterministic stack automata which can read the stack in only one direction is provided by a restricted class of indexed languages. Several of the fundamental properties of the classes of languages defined by nested stack automata are also presented.

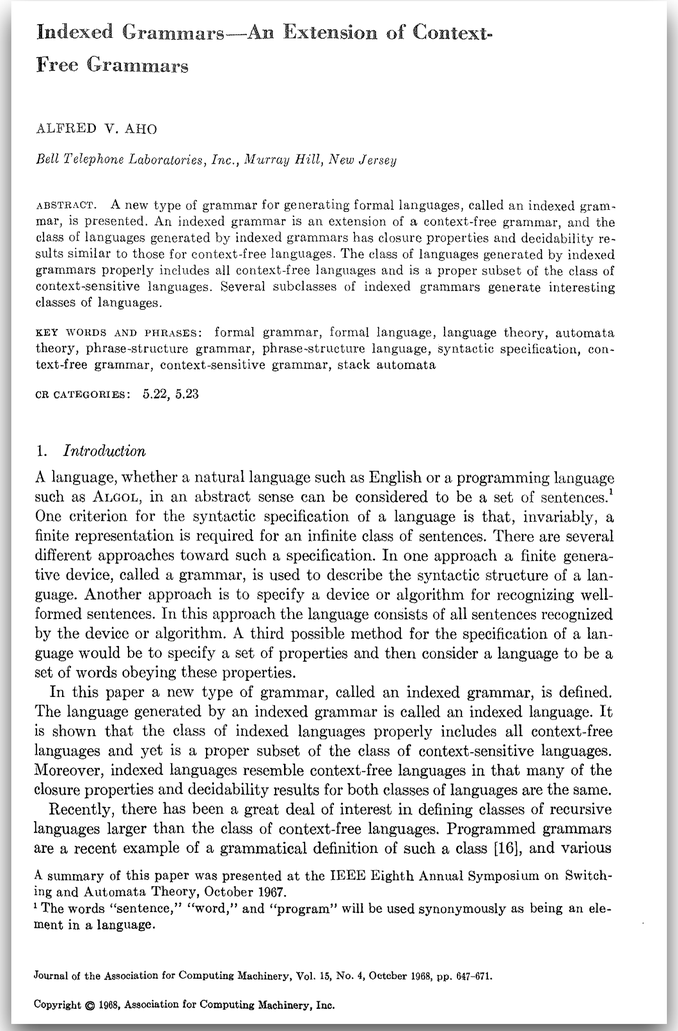
Recently, another automaton model, called a list-storage acceptor (lsa), was introduced [9]. The lsa is an automaton model with the auxiliary memory organized as a linear list. The unrestricted lsa is capable of defining exactly all the recursively enumerable sets. A temporal restriction on the input operation of lsa is used to limit the definitional power of lsa to a subset of the recursive sets. A nested stack automaton, on the other hand, achieves recursiveness by means of a spatial restriction on the access of information from the memory structure.

Journal of the Association for Computing Machinery, Vol. 16, No. 3, July 1969, pp. 383-406.

PhD work

Alfred Aho

- Introduced Indexed Grammar (IG) is a generator for a class of languages called Indexed Languages.
  - More expressive than context-free grammars (CFG).
  - IG has similar closure and decidability results to CFG, including a pumping lemma (by Takeshi Hayashi, 1973).
  - IG is a proper subset of context-sensitive grammars.
- Structurally, an index grammar has an additional component, wrt. CFG, called flags.
- Essentially, a flag further expands a non-terminal when it occurs right after it in a derivation.
- Flag symbols distribute over non-terminals or terminals in a derivation.
- It is a way to “type" productions in a grammar.



Indexed Grammars—An Extension of Context-Free Grammars

ALFRED V. AHO

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. A new type of grammar for generating formal languages, called an indexed grammar, is presented. An indexed grammar is an extension of a context-free grammar, and the class of languages generated by indexed grammars has closure properties and decidability results similar to those for context-free languages. The class of languages generated by indexed grammars properly includes all context-free languages and is a proper subset of the class of context-sensitive languages. Several subclasses of indexed grammars generate interesting classes of languages.

KEY WORDS AND PHRASES: formal grammar, formal language, language theory, automata theory, phrase-structure grammar, phrase-structure language, syntactic specification, context-free grammar, context-sensitive grammar, stack automata

CR CATEGORIES: 5.22, 5.23

1. *Introduction*

A language, whether a natural language such as English or a programming language such as ALGOL, in an abstract sense can be considered to be a set of sentences.[1] One criterion for the syntactic specification of a language is that, invariably, a finite representation is required for an infinite class of sentences. There are several different approaches toward such a specification. In one approach a finite generative device, called a grammar, is used to describe the syntactic structure of a language. Another approach is to specify a device or algorithm for recognizing well-formed sentences. In this approach the language consists of all sentences recognized by the device or algorithm. A third possible method for the specification of a language would be to specify a set of properties and then consider a language to be a set of words obeying these properties.

In this paper a new type of grammar, called an indexed grammar, is defined. The language generated by an indexed grammar is called an indexed language. It is shown that the class of indexed languages properly includes all context-free languages and yet is a proper subset of the class of context-sensitive languages. Moreover, indexed languages resemble context-free languages in that many of the closure properties and decidability results for both classes of languages are the same.

Recently, there has been a great deal of interest in defining classes of recursive languages larger than the class of context-free languages. Programmed grammars are a recent example of a grammatical definition of such a class [16], and various

A summary of this paper was presented at the IEEE Eighth Annual Symposium on Switching and Automata Theory, October 1967.

[1] The words “sentence,” “word,” and “program” will be used synonymously as being an element in a language.

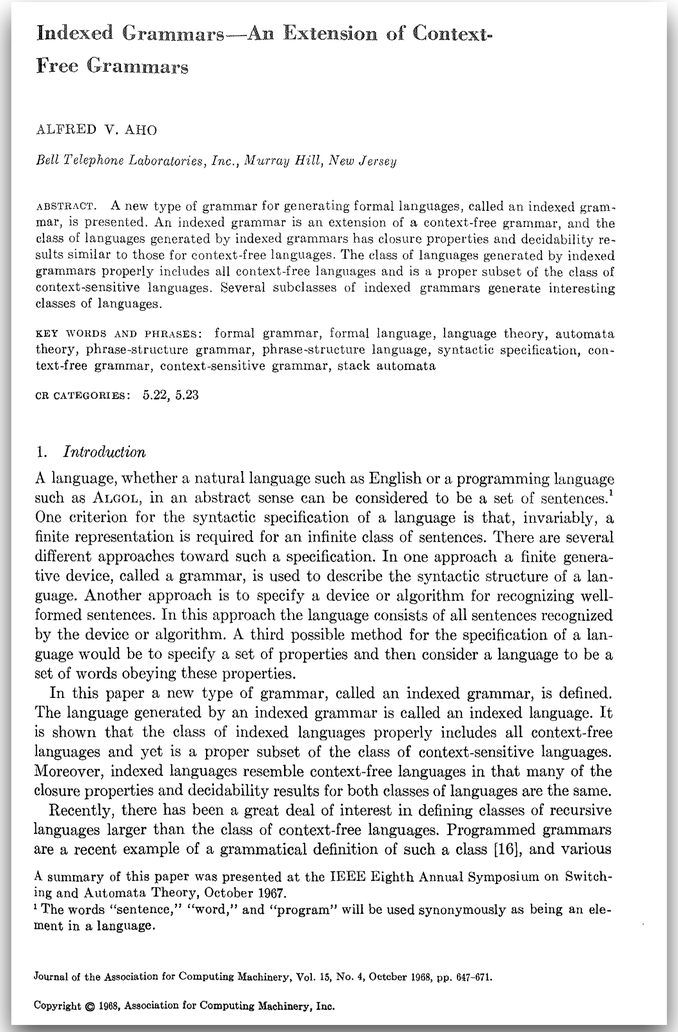
Journal of the Association for Computing Machinery, Vol. 15, No. 4, October 1968, pp. 647–671.

Aho, A. V. (1968). Indexed Grammars - an extension of context-free grammars. Journal of the ACM, 15(4), pp. 647-671. https://doi.org/10.1145/321479.321488

PhD work

Alfred Aho

- Indexed Grammar example:
  - $L = \{a^n b^n c^n \mid n \geq 1\}$,
  - $P = \{S \rightarrow aAfc,\ A \rightarrow aAgc,\ A \rightarrow B\}$, is the set of productions,
  - $F = \{f = [B \rightarrow b],\ g = [B \rightarrow bB]\}$, is the set of flags.
  - $S \Rightarrow aAfc \Rightarrow aaAgcfc = aaAgfcc \Rightarrow aaBgfcc \Rightarrow aabBfcc \Rightarrow aabbcc$.
  - Here *f* was distributed over the (singleton) list of indexed non-terminals *[Ag]*, after *A* was replaced by *aAgc*.
- *L* is not in CFL, by the pumping lemma.



Indexed Grammars—An Extension of Context-Free Grammars

ALFRED V. AHO

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. A new type of grammar for generating formal languages, called an indexed grammar, is presented. An indexed grammar is an extension of a context-free grammar, and the class of languages generated by indexed grammars has closure properties and decidability results similar to those for context-free languages. The class of languages generated by indexed grammars properly includes all context-free languages and is a proper subset of the class of context-sensitive languages. Several subclasses of indexed grammars generate interesting classes of languages.

KEY WORDS AND PHRASES: formal grammar, formal language, language theory, automata theory, phrase-structure grammar, phrase-structure language, syntactic specification, context-free grammar, context-sensitive grammar, stack automata

CR CATEGORIES: 5.22, 5.23

1. *Introduction*

A language, whether a natural language such as English or a programming language such as ALGOL, in an abstract sense can be considered to be a set of sentences.[1] One criterion for the syntactic specification of a language is that, invariably, a finite representation is required for an infinite class of sentences. There are several different approaches toward such a specification. In one approach a finite generative device, called a grammar, is used to describe the syntactic structure of a language. Another approach is to specify a device or algorithm for recognizing well-formed sentences. In this approach the language consists of all sentences recognized by the device or algorithm. A third possible method for the specification of a language would be to specify a set of properties and then consider a language to be a set of words obeying these properties.

In this paper a new type of grammar, called an indexed grammar, is defined. The language generated by an indexed grammar is called an indexed language. It is shown that the class of indexed languages properly includes all context-free languages and yet is a proper subset of the class of context-sensitive languages. Moreover, indexed languages resemble context-free languages in that many of the closure properties and decidability results for both classes of languages are the same.

Recently, there has been a great deal of interest in defining classes of recursive languages larger than the class of context-free languages. Programmed grammars are a recent example of a grammatical definition of such a class [16], and various

A summary of this paper was presented at the IEEE Eighth Annual Symposium on Switching and Automata Theory, October 1967.

[1] The words "sentence," "word," and "program" will be used synonymously as being an element in a language.

Journal of the Association for Computing Machinery, Vol. 15, No. 4, October 1968, pp. 647-671.

Aho, A. V. (1968). Indexed Grammars - an extension of context-free grammars. Journal of the ACM, 15(4), pp. 647-671. https://doi.org/10.1145/321479.321488

PhD work

Alfred Aho

- Nested stack automata (NSA) are recognizers for indexed languages.
- They are a generalization of stack automata (SA), devices similar to pushdown automata that may move up and down the stack, or remain stationary.
- NSA generalizes SA by allowing the possibility to create a stack, possibly embedded into another.
- The list of symbols

  $L_1 = a_1\ a_1\ L_2\ a_3\ a_4\ L_3\ a_5$

  $L_2 = b_1\ b_2$

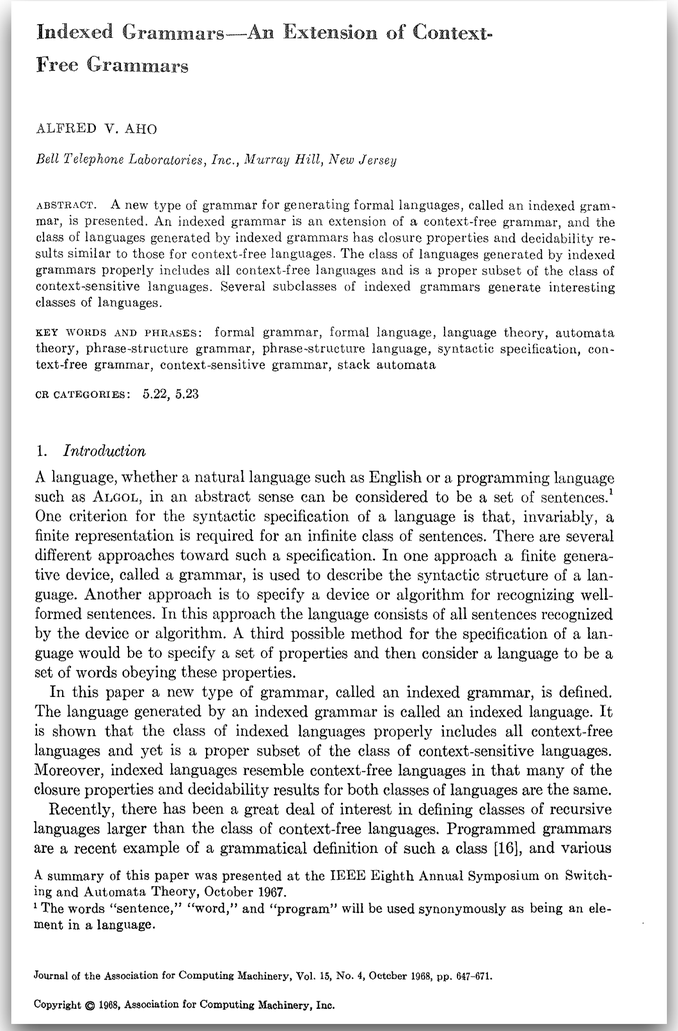
  $L_3 = c_1 L_4\ c_2$

  $L_4 = d_1$

  would be represented in an NSA (working) tape as follows.

  $\$a_1\ a_2\ \$b_1\ b_2\ /\ a_3\ a_4\ \$c_1\ \$d_1\ /\ c_2\ /\ a_5\ \#$

- NSA operates under pushdown mode, read mode or stack creation mode.



Indexed Grammars—An Extension of Context-Free Grammars

ALFRED V. AHO

Bell Telephone Laboratories, Inc., Murray Hill, New Jersey

ABSTRACT. A new type of grammar for generating formal languages, called an indexed grammar, is presented. An indexed grammar is an extension of a context-free grammar, and the class of languages generated by indexed grammars has closure properties and decidability results similar to those for context-free languages. The class of languages generated by indexed grammars properly includes all context-free languages and is a proper subset of the class of context-sensitive languages. Several subclasses of indexed grammars generate interesting classes of languages.

KEY WORDS AND PHRASES: formal grammar, formal language, language theory, automata theory, phrase-structure grammar, phrase-structure language, syntactic specification, context-free grammar, context-sensitive grammar, stack automata

CR CATEGORIES: 5.22, 5.23

1. *Introduction*

A language, whether a natural language such as English or a programming language such as ALGOL, in an abstract sense can be considered to be a set of sentences.[1] One criterion for the syntactic specification of a language is that, invariably, a finite representation is required for an infinite class of sentences. There are several different approaches toward such a specification. In one approach a finite generative device, called a grammar, is used to describe the syntactic structure of a language. Another approach is to specify a device or algorithm for recognizing well-formed sentences. In this approach the language consists of all sentences recognized by the device or algorithm. A third possible method for the specification of a language would be to specify a set of properties and then consider a language to be a set of words obeying these properties.

In this paper a new type of grammar, called an indexed grammar, is defined. The language generated by an indexed grammar is called an indexed language. It is shown that the class of indexed languages properly includes all context-free languages and yet is a proper subset of the class of context-sensitive languages. Moreover, indexed languages resemble context-free languages in that many of the closure properties and decidability results for both classes of languages are the same.

Recently, there has been a great deal of interest in defining classes of recursive languages larger than the class of context-free languages. Programmed grammars are a recent example of a grammatical definition of such a class [16], and various

A summary of this paper was presented at the IEEE Eighth Annual Symposium on Switching and Automata Theory, October 1967.

[1] The words "sentence," "word," and "program" will be used synonymously as being an element in a language.

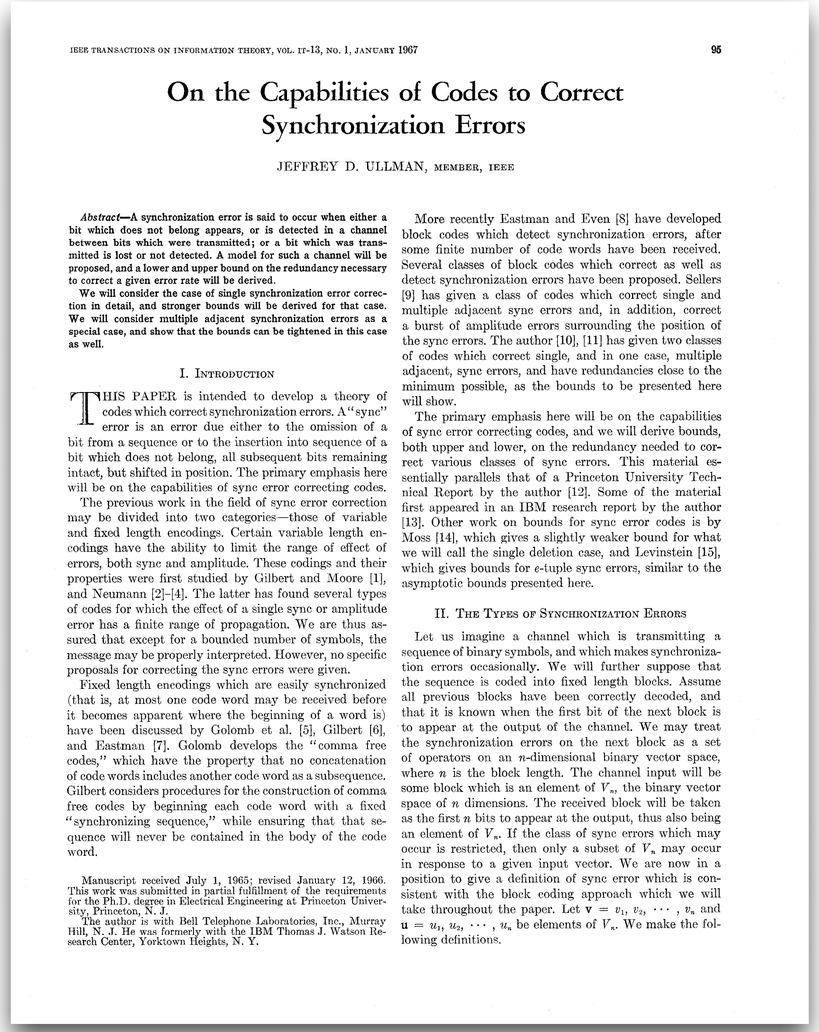
Journal of the Association for Computing Machinery, Vol. 15, No. 4, October 1968, pp. 647–671.

Aho, A. V. (1968). Indexed Grammars - an extension of context-free grammars. Journal of the ACM, 15(4), pp. 647-671. https://doi.org/10.1145/321479.321488

PhD work

Jeffrey Ullman

- Synchronization Error Correcting Codes
- Princeton, 1966
- Supervisors: Arthur Jay Bernstein and Archie Charles McKellar



On the Capabilities of Codes to Correct Synchronization Errors

JEFFREY D. ULLMAN, MEMBER, IEEE

Abstract—**A synchronization error is said to occur when either a bit which does not belong appears, or is detected in a channel between bits which were transmitted; or a bit which was transmitted is lost or not detected. A model for such a channel will be proposed, and a lower and upper bound on the redundancy necessary to correct a given error rate will be derived.**

We will consider the case of single synchronization error correction in detail, and stronger bounds will be derived for that case. We will consider multiple adjacent synchronization errors as a special case, and show that the bounds can be tightened in this case as well.

I. Introduction

THIS PAPER is intended to develop a theory of codes which correct synchronization errors. A "sync" error is an error due either to the omission of a bit from a sequence or to the insertion into sequence of a bit which does not belong, all subsequent bits remaining intact, but shifted in position. The primary emphasis here will be on the capabilities of sync error correcting codes.

The previous work in the field of sync error correction may be divided into two categories—those of variable and fixed length encodings. Certain variable length encodings have the ability to limit the range of effect of errors, both sync and amplitude. These codings and their properties were first studied by Gilbert and Moore [1], and Neumann [2]–[4]. The latter has found several types of codes for which the effect of a single sync or amplitude error has a finite range of propagation. We are thus assured that except for a bounded number of symbols, the message may be properly interpreted. However, no specific proposals for correcting the sync errors were given.

Fixed length encodings which are easily synchronized (that is, at most one code word may be received before it becomes apparent where the beginning of a word is) have been discussed by Golomb et al. [5], Gilbert [6], and Eastman [7]. Golomb develops the "comma free codes," which have the property that no concatenation of code words includes another code word as a subsequence. Gilbert considers procedures for the construction of comma free codes by beginning each code word with a fixed "synchronizing sequence," while ensuring that that sequence will never be contained in the body of the code word.

Manuscript received July 1, 1965; revised January 12, 1966. This work was submitted in partial fulfillment of the requirements for the Ph.D. degree in Electrical Engineering at Princeton University, Princeton, N. J.

The author is with Bell Telephone Laboratories, Inc., Murray Hill, N. J. He was formerly with the IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y.

More recently Eastman and Even [8] have developed block codes which detect synchronization errors, after some finite number of code words have been received. Several classes of block codes which correct as well as detect synchronization errors have been proposed. Sellers [9] has given a class of codes which correct single and multiple adjacent sync errors and, in addition, correct a burst of amplitude errors surrounding the position of the sync errors. The author [10], [11] has given two classes of codes which correct single, and in one case, multiple adjacent, sync errors, and have redundancies close to the minimum possible, as the bounds to be presented here will show.

The primary emphasis here will be on the capabilities of sync error correcting codes, and we will derive bounds, both upper and lower, on the redundancy needed to correct various classes of sync errors. This material essentially parallels that of a Princeton University Technical Report by the author [12]. Some of the material first appeared in an IBM research report by the author [13]. Other work on bounds for sync error codes is by Moss [14], which gives a slightly weaker bound for what we will call the single deletion case, and Levinstein [15], which gives bounds for e-tuple sync errors, similar to the asymptotic bounds presented here.

II. The Types of Synchronization Errors

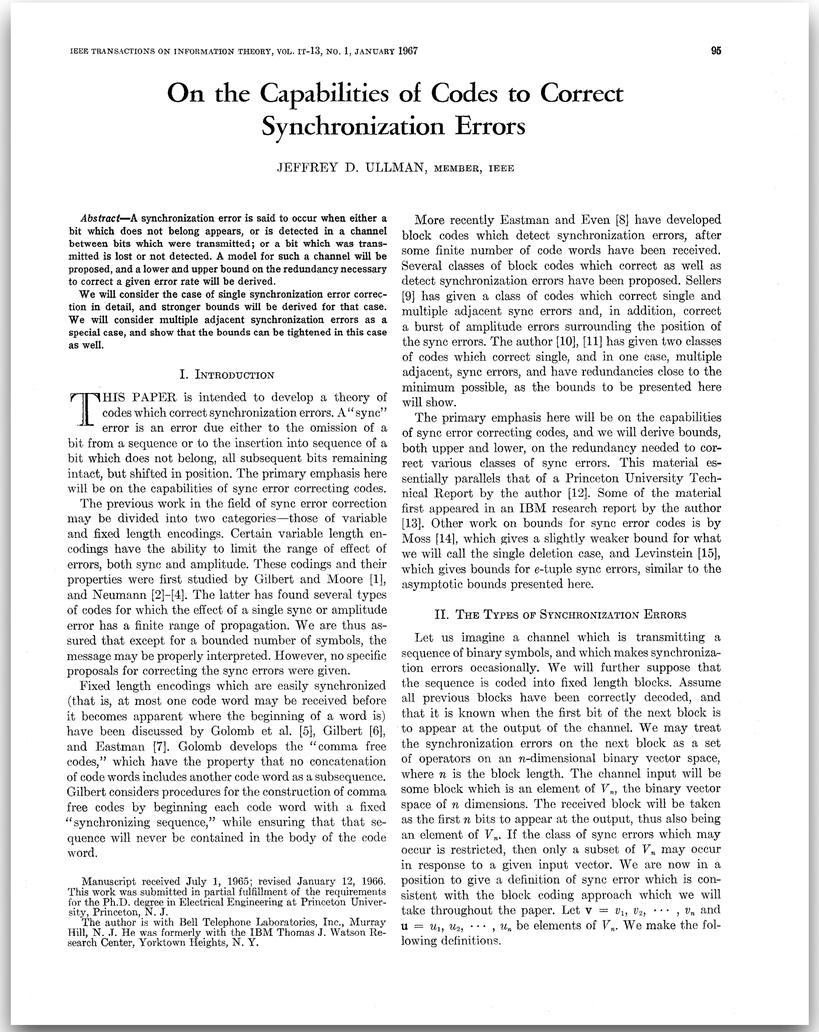
Let us imagine a channel which is transmitting a sequence of binary symbols, and which makes synchronization errors occasionally. We will further suppose that the sequence is coded into fixed length blocks. Assume all previous blocks have been correctly decoded, and that it is known when the first bit of the next block is to appear at the output of the channel. We may treat the synchronization errors on the next block as a set of operators on an n-dimensional binary vector space, where n is the block length. The channel input will be some block which is an element of V_n, the binary vector space of n dimensions. The received block will be taken as the first n bits to appear at the output, thus also being an element of V_n. If the class of sync errors which may occur is restricted, then only a subset of V_n may occur in response to a given input vector. We are now in a position to give a definition of sync error which is consistent with the block coding approach which we will take throughout the paper. Let $\mathbf{v} = v_1, v_2, \cdots, v_n$ and $\mathbf{u} = u_1, u_2, \cdots, u_n$ be elements of V_n. We make the following definitions.

Ullman, J. (1967). On the capabilities of codes to correct synchronization errors. IEEE Transactions on Information Theory, 13(1), 95–105. doi:10.1109/tit.1967.1053954

PhD work

Jeffrey Ullman

- Intends to develop a theory of codes that correct synchronization errors (absence or inclusion of a bit) in a stream of binary symbols.
- A sequence of bits is encoded as a fixed-length block.
- Synchronization errors on a block are a set of operators on an *n*-dimensional binary vector space, where *n* is the block length.
- Bounds on the necessary redundancy to correct a given error rate are derived on a model for such a stream.



IEEE TRANSACTIONS ON INFORMATION THEORY, VOL. IT-13, NO. 1, JANUARY 1967

On the Capabilities of Codes to Correct Synchronization Errors

JEFFREY D. ULLMAN, MEMBER, IEEE

***Abstract*—A synchronization error is said to occur when either a bit which does not belong appears, or is detected in a channel between bits which were transmitted; or a bit which was transmitted is lost or not detected. A model for such a channel will be proposed, and a lower and upper bound on the redundancy necessary to correct a given error rate will be derived.**

We will consider the case of single synchronization error correction in detail, and stronger bounds will be derived for that case. We will consider multiple adjacent synchronization errors as a special case, and show that the bounds can be tightened in this case as well.

I. INTRODUCTION

THIS PAPER is intended to develop a theory of codes which correct synchronization errors. A "sync" error is an error due either to the omission of a bit from a sequence or to the insertion into sequence of a bit which does not belong, all subsequent bits remaining intact, but shifted in position. The primary emphasis here will be on the capabilities of sync error correcting codes.

The previous work in the field of sync error correction may be divided into two categories—those of variable and fixed length encodings. Certain variable length encodings have the ability to limit the range of effect of errors, both sync and amplitude. These codings and their properties were first studied by Gilbert and Moore [1], and Neumann [2]–[4]. The latter has found several types of codes for which the effect of a single sync or amplitude error has a finite range of propagation. We are thus assured that except for a bounded number of symbols, the message may be properly interpreted. However, no specific proposals for correcting the sync errors were given.

Fixed length encodings which are easily synchronized (that is, at most one code word may be received before it becomes apparent where the beginning of a word is) have been discussed by Golomb et al. [5], Gilbert [6], and Eastman [7]. Golomb develops the "comma free codes," which have the property that no concatenation of code words includes another code word as a subsequence. Gilbert considers procedures for the construction of comma free codes by beginning each code word with a fixed "synchronizing sequence," while ensuring that that sequence will never be contained in the body of the code word.

Manuscript received July 1, 1965; revised January 12, 1966. This work was submitted in partial fulfillment of the requirements for the Ph.D. degree in Electrical Engineering at Princeton University, Princeton, N. J.

The author is with Bell Telephone Laboratories, Inc., Murray Hill, N. J. He was formerly with the IBM Thomas J. Watson Research Center, Yorktown Heights, N. Y.

More recently Eastman and Even [8] have developed block codes which detect synchronization errors, after some finite number of code words have been received. Several classes of block codes which correct as well as detect synchronization errors have been proposed. Sellers [9] has given a class of codes which correct single and multiple adjacent sync errors and, in addition, correct a burst of amplitude errors surrounding the position of the sync errors. The author [10], [11] has given two classes of codes which correct single, and in one case, multiple adjacent, sync errors, and have redundancies close to the minimum possible, as the bounds to be presented here will show.

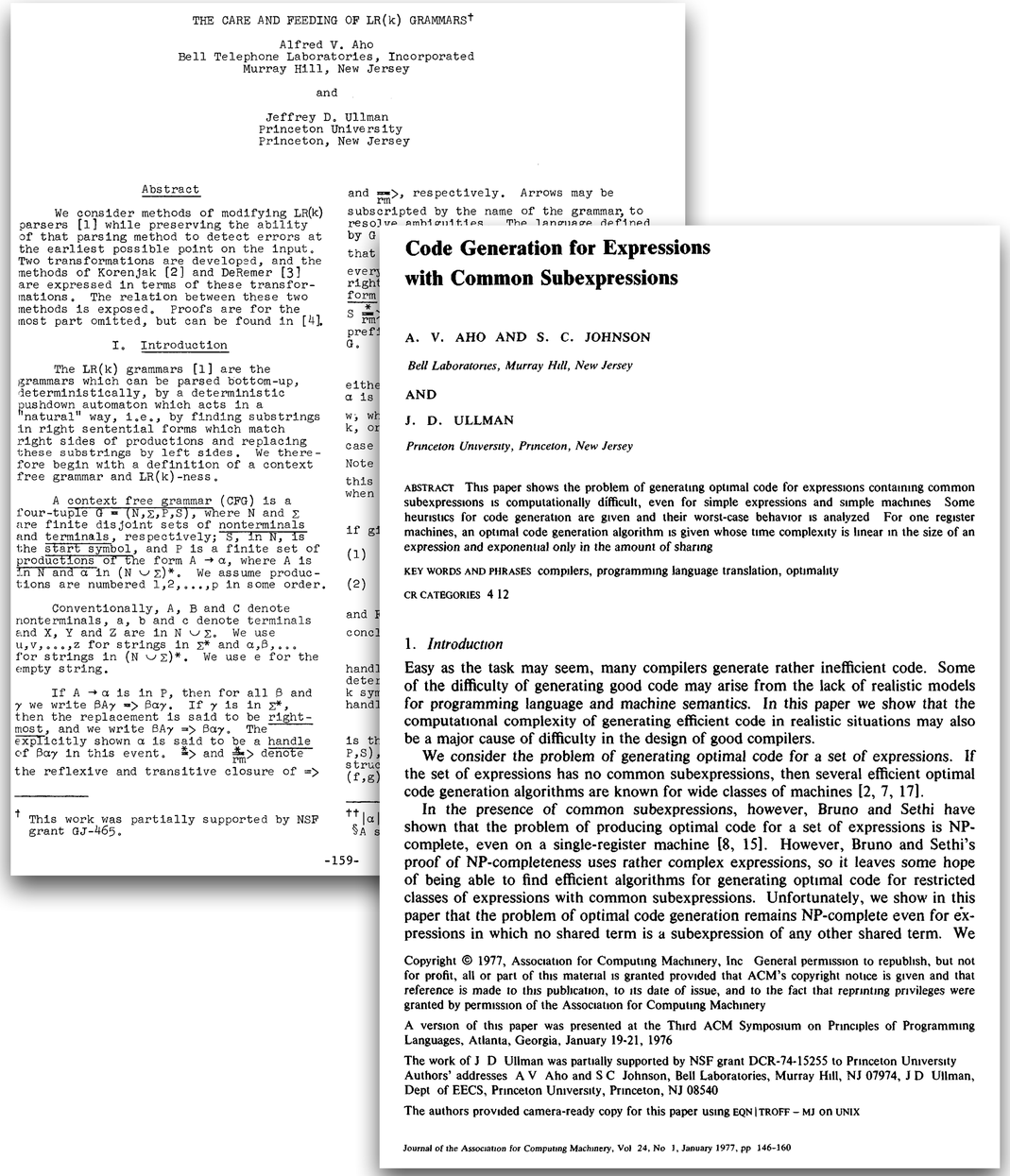
The primary emphasis here will be on the capabilities of sync error correcting codes, and we will derive bounds, both upper and lower, on the redundancy needed to correct various classes of sync errors. This material essentially parallels that of a Princeton University Technical Report by the author [12]. Some of the material first appeared in an IBM research report by the author [13]. Other work on bounds for sync error codes is by Moss [14], which gives a slightly weaker bound for what we will call the single deletion case, and Levinstein [15], which gives bounds for e-tuple sync errors, similar to the asymptotic bounds presented here.

II. THE TYPES OF SYNCHRONIZATION ERRORS

Let us imagine a channel which is transmitting a sequence of binary symbols, and which makes synchronization errors occasionally. We will further suppose that the sequence is coded into fixed length blocks. Assume all previous blocks have been correctly decoded, and that it is known when the first bit of the next block is to appear at the output of the channel. We may treat the synchronization errors on the next block as a set of operators on an n-dimensional binary vector space, where n is the block length. The channel input will be some block which is an element of V_n, the binary vector space of n dimensions. The received block will be taken as the first n bits to appear at the output, thus also being an element of V_n. If the class of sync errors which may occur is restricted, then only a subset of V_n may occur in response to a given input vector. We are now in a position to give a definition of sync error which is consistent with the block coding approach which we will take throughout the paper. Let $\mathbf{v} = v_1, v_2, \cdots, v_n$ and $\mathbf{u} = u_1, u_2, \cdots, u_n$ be elements of V_n. We make the following definitions.

Ullman, J. (1967). On the capabilities of codes to correct synchronization errors. IEEE Transactions on Information Theory, 13(1), 95–105. doi:10.1109/tit.1967.1053954

Two papers



THE CARE AND FEEDING OF LR(k) GRAMMARS†

Alfred V. Aho
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

and

Jeffrey D. Ullman
Princeton University
Princeton, New Jersey

Abstract

We consider methods of modifying LR(k) parsers [1] while preserving the ability of that parsing method to detect errors at the earliest possible point on the input. Two transformations are developed, and the methods of Korenjak [2] and DeRemer [3] are expressed in terms of these transformations. The relation between these two methods is exposed. Proofs are for the most part omitted, but can be found in [4].

I. Introduction

The LR(k) grammars [1] are the grammars which can be parsed bottom-up, deterministically, by a deterministic pushdown automaton which acts in a "natural" way, i.e., by finding substrings in right sentential forms which match right sides of productions and replacing these substrings by left sides. We therefore begin with a definition of a context free grammar and LR(k)-ness.

A context free grammar (CFG) is a four-tuple $G = (N,\Sigma,P,S)$, where N and Σ are finite disjoint sets of nonterminals and terminals, respectively; S, in N, is the start symbol, and P is a finite set of productions of the form $A \to \alpha$, where A is in N and α in $(N \cup \Sigma)^*$. We assume productions are numbered 1,2,...,p in some order.

Conventionally, A, B and C denote nonterminals, a, b and c denote terminals and X, Y and Z are in $N \cup \Sigma$. We use u,v,...,z for strings in Σ^* and $\alpha,\beta,\ldots$ for strings in $(N \cup \Sigma)^*$. We use e for the empty string.

If $A \to \alpha$ is in P, then for all β and γ we write $\beta A\gamma \Rightarrow \beta\alpha\gamma$. If γ is in Σ^*, then the replacement is said to be rightmost, and we write $\beta A\gamma \Rightarrow_{rm} \beta\alpha\gamma$. The explicitly shown α is said to be a handle of $\beta\alpha\gamma$ in this event. $\overset{*}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ denote the reflexive and transitive closure of $\Rightarrow$ and $\underset{rm}{\Rightarrow}$, respectively. Arrows may be subscripted by the name of the grammar to resolve ambiguities. The language defined by G

† This work was partially supported by NSF grant GJ-465.

Code Generation for Expressions with Common Subexpressions

A. V. AHO AND S. C. JOHNSON

Bell Laboratories, Murray Hill, New Jersey

AND

J. D. ULLMAN

Princeton University, Princeton, New Jersey

ABSTRACT This paper shows the problem of generating optimal code for expressions containing common subexpressions is computationally difficult, even for simple expressions and simple machines Some heuristics for code generation are given and their worst-case behavior is analyzed For one register machines, an optimal code generation algorithm is given whose time complexity is linear in the size of an expression and exponential only in the amount of sharing

KEY WORDS AND PHRASES compilers, programming language translation, optimality

CR CATEGORIES 4 12

1. *Introduction*

Easy as the task may seem, many compilers generate rather inefficient code. Some of the difficulty of generating good code may arise from the lack of realistic models for programming language and machine semantics. In this paper we show that the computational complexity of generating efficient code in realistic situations may also be a major cause of difficulty in the design of good compilers.

We consider the problem of generating optimal code for a set of expressions. If the set of expressions has no common subexpressions, then several efficient optimal code generation algorithms are known for wide classes of machines [2, 7, 17].

In the presence of common subexpressions, however, Bruno and Sethi have shown that the problem of producing optimal code for a set of expressions is NP-complete, even on a single-register machine [8, 15]. However, Bruno and Sethi's proof of NP-completeness uses rather complex expressions, so it leaves some hope of being able to find efficient algorithms for generating optimal code for restricted classes of expressions with common subexpressions. Unfortunately, we show in this paper that the problem of optimal code generation remains NP-complete even for expressions in which no shared term is a subexpression of any other shared term. We

A version of this paper was presented at the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, January 19-21, 1976

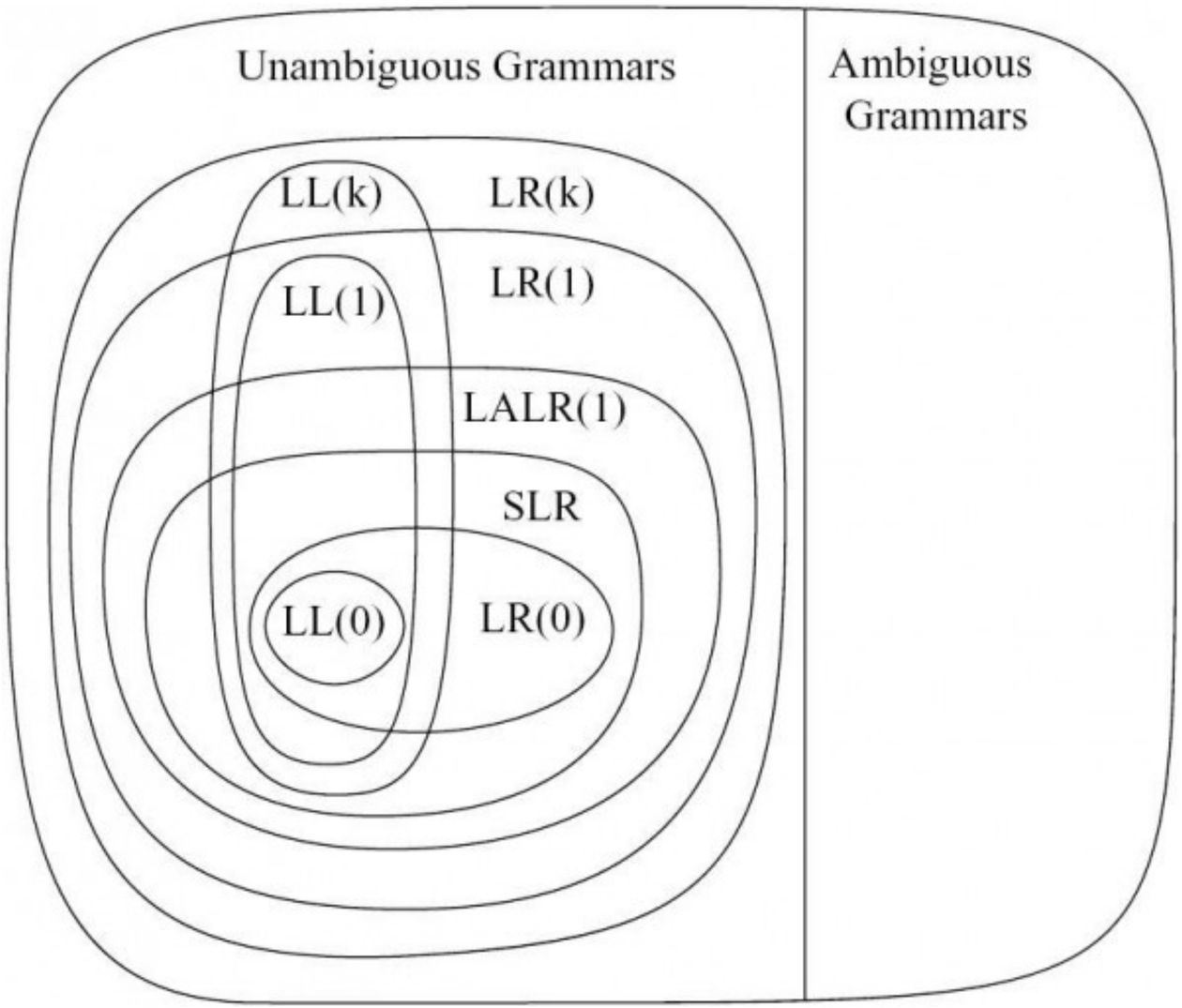
The work of J D Ullman was partially supported by NSF grant DCR-74-15255 to Princeton University
Authors' addresses A V Aho and S C Johnson, Bell Laboratories, Murray Hill, NJ 07974, J D Ullman, Dept of EECS, Princeton University, Princeton, NJ 08540

The authors provided camera-ready copy for this paper using EQN | TROFF – MJ on UNIX

Journal of the Association for Computing Machinery, Vol 24, No 1, January 1977, pp 146-160

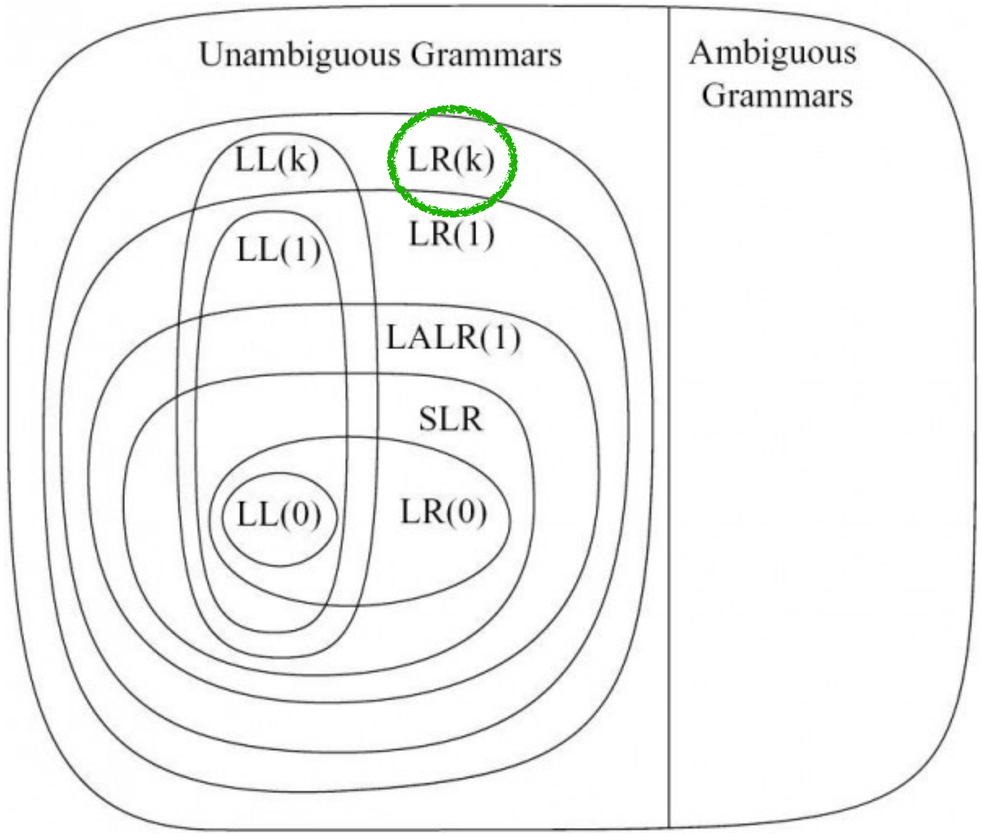
(Intermezzo)

Context-free parsing grammars



(Intermezzo)

Context-free parsing grammars



INFORMATION AND CONTROL 8, 607–639 (1965)

On the Translation of Languages from Left to Right

DONALD E. KNUTH

Mathematics Department, California Institute of Technology, Pasadena, California

There has been much recent interest in languages whose grammar is sufficiently simple that an efficient left-to-right parsing algorithm can be mechanically produced from the grammar. In this paper, we define LR(k) grammars, which are perhaps the most general ones of this type, and they provide the basis for understanding all of the special tricks which have been used in the construction of parsing algorithms for languages with simple structure, e.g. algebraic languages. We give algorithms for deciding if a given grammar satisfies the LR(k) condition, for given k, and also give methods for generating recognizers for LR(k) grammars. It is shown that the problem of whether or not a grammar is LR(k) for *some* k is undecidable, and the paper concludes by establishing various connections between LR(k) grammars and deterministic languages. In particular, the LR(k) condition is a natural analogue, for grammars, of the deterministic condition, for languages.

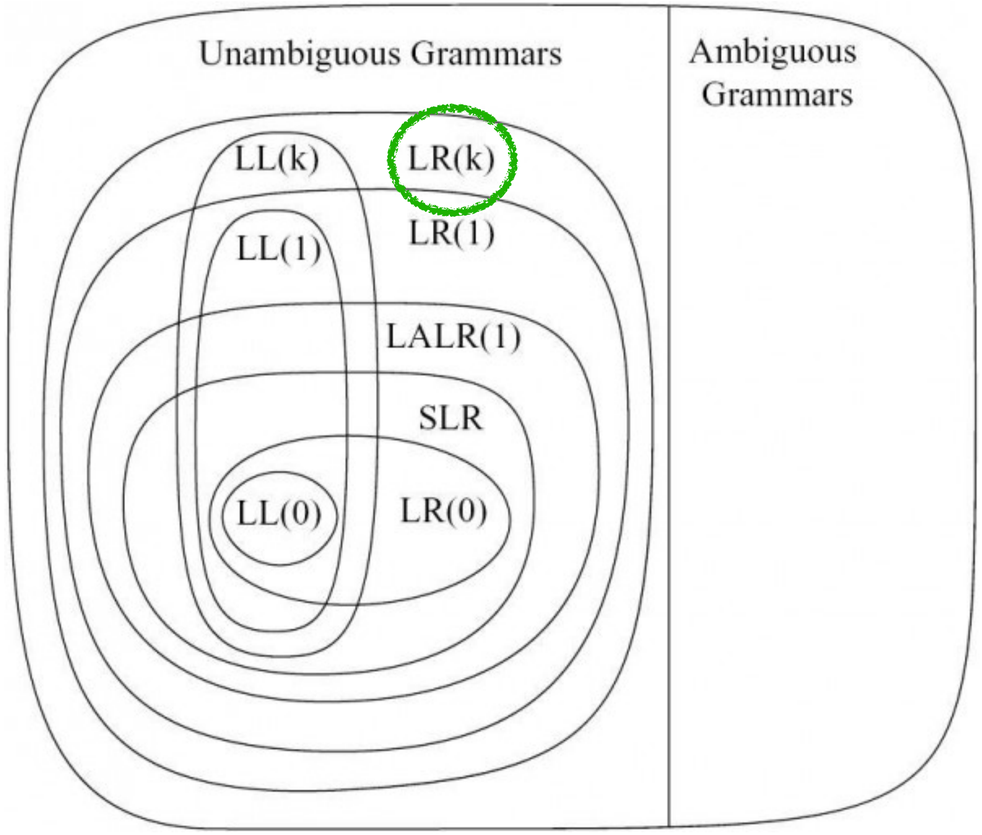
I. INTRODUCTION AND DEFINITIONS

The word "language" will be used here to denote a set of character strings which has been variously called a *context free language*, a (*simple*) *phrase structure language*, a *constituent-structure language*, a *definable set*, a *BNF language*, a *Chomsky type 2* (*or type 4*) *language*, a *push-down automaton language*, etc. Such languages have aroused wide interest because they serve as approximate models for natural languages and computer programming languages, among others. In this paper we single out an important class of languages which will be called *translatable from left to right*; this means if we read the characters of a string from left to right, and look a given finite number of characters ahead, we are able to parse the given string without ever backing up to consider a previous decision. Such languages are particularly important in the case of computer programming, since this condition means a parsing algorithm can be mechanically constructed which requires an execution time at worst proportional to the length of the string being parsed. Special-purpose

D. Knuth, On the translation of languages from left to right, Information and Control 8, 607-639 (1965) https://doi.org/10.1016/S0019-9958(65)90426-2

(Intermezzo)

Context-free parsing grammars



- Produces parsing tables with 10^3 states for an ALGOL-like programming language.

INFORMATION AND CONTROL 8, 607–639 (1965)

On the Translation of Languages from Left to Right

DONALD E. KNUTH

Mathematics Department, California Institute of Technology, Pasadena, California

There has been much recent interest in languages whose grammar is sufficiently simple that an efficient left-to-right parsing algorithm can be mechanically produced from the grammar. In this paper, we define LR(k) grammars, which are perhaps the most general ones of this type, and they provide the basis for understanding all of the special tricks which have been used in the construction of parsing algorithms for languages with simple structure, e.g. algebraic languages. We give algorithms for deciding if a given grammar satisfies the LR(k) condition, for given k, and also give methods for generating recognizers for LR(k) grammars. It is shown that the problem of whether or not a grammar is LR(k) for *some* k is undecidable, and the paper concludes by establishing various connections between LR(k) grammars and deterministic languages. In particular, the LR(k) condition is a natural analogue, for grammars, of the deterministic condition, for languages.

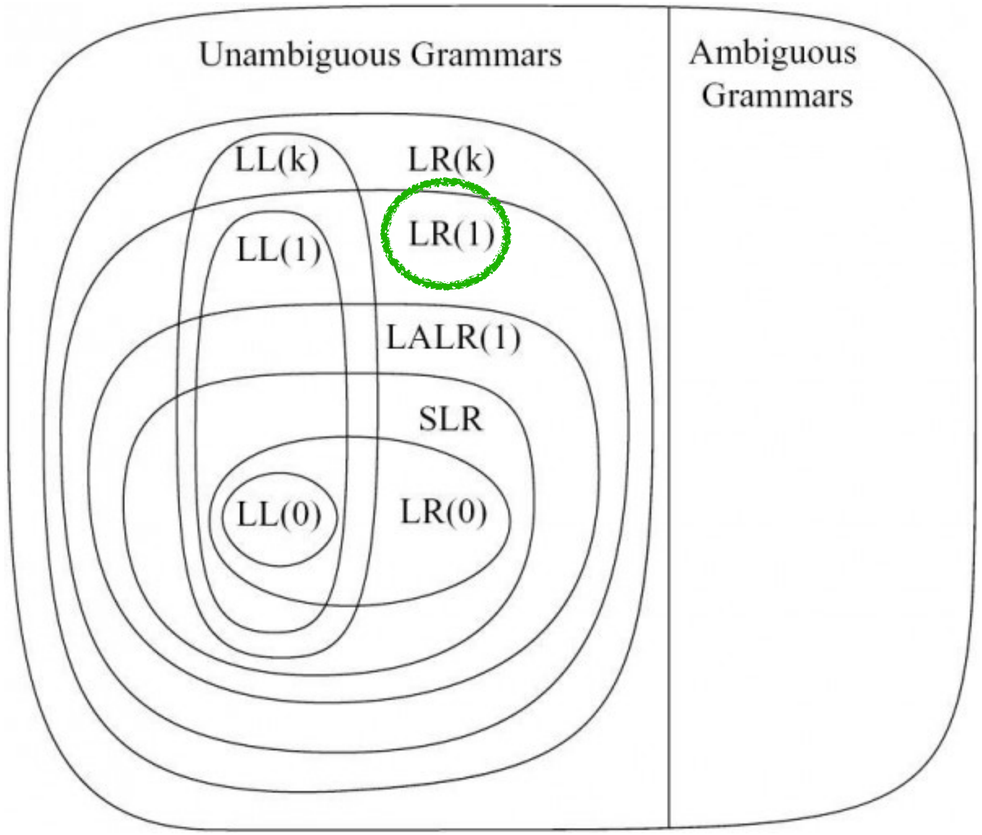
I. INTRODUCTION AND DEFINITIONS

The word "language" will be used here to denote a set of character strings which has been variously called a *context free language*, a (*simple*) *phrase structure language*, a *constituent-structure language*, a *definable set*, a *BNF language*, a *Chomsky type 2* (*or type 4*) *language*, a *push-down automaton language*, etc. Such languages have aroused wide interest because they serve as approximate models for natural languages and computer programming languages, among others. In this paper we single out an important class of languages which will be called *translatable from left to right*; this means if we read the characters of a string from left to right, and look a given finite number of characters ahead, we are able to parse the given string without ever backing up to consider a previous decision. Such languages are particularly important in the case of computer programming, since this condition means a parsing algorithm can be mechanically constructed which requires an execution time at worst proportional to the length of the string being parsed. Special-purpose

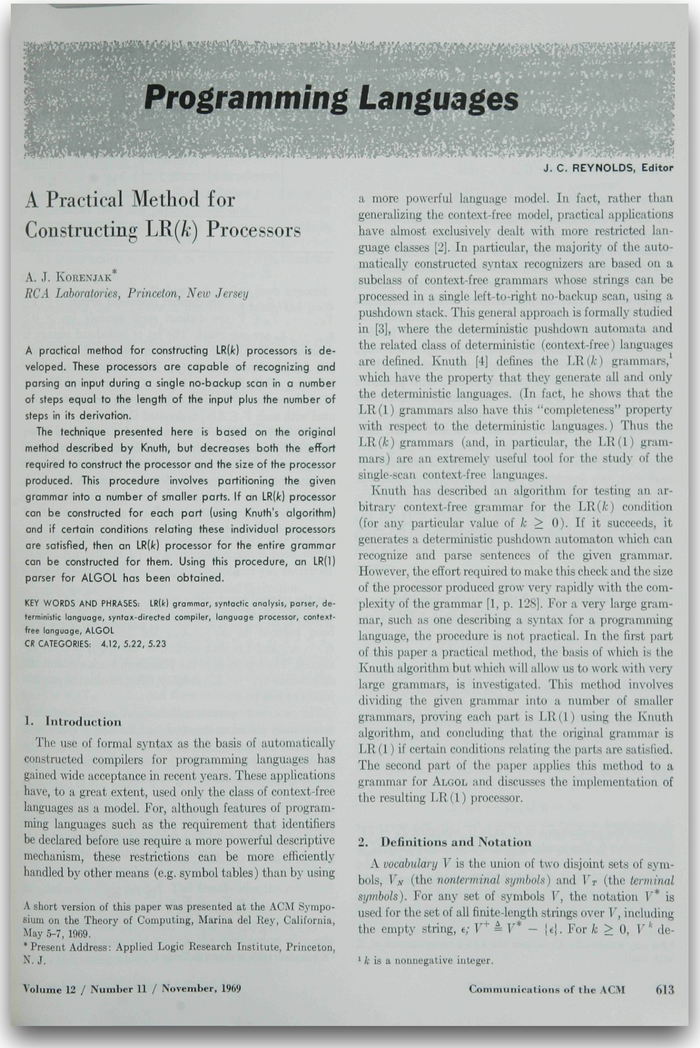
D. Knuth, On the translation of languages from left to right, Information and Control 8, 607-639 (1965) https://doi.org/10.1016/S0019-9958(65)90426-2

(Intermezzo)

Context-free parsing grammars



- Proposes partitioning the grammar to generate efficient LR(1) parsers.



Programming Languages

J. C. REYNOLDS, Editor

A Practical Method for Constructing LR(k) Processors

A. J. KORENJAK*
RCA Laboratories, Princeton, New Jersey

A practical method for constructing LR(k) processors is developed. These processors are capable of recognizing and parsing an input during a single no-backup scan in a number of steps equal to the length of the input plus the number of steps in its derivation.

The technique presented here is based on the original method described by Knuth, but decreases both the effort required to construct the processor and the size of the processor produced. This procedure involves partitioning the given grammar into a number of smaller parts. If an LR(k) processor can be constructed for each part (using Knuth's algorithm) and if certain conditions relating these individual processors are satisfied, then an LR(k) processor for the entire grammar can be constructed for them. Using this procedure, an LR(1) parser for ALGOL has been obtained.

KEY WORDS AND PHRASES: LR(k) grammar, syntactic analysis, parser, deterministic language, syntax-directed compiler, language processor, context-free language, ALGOL
CR CATEGORIES: 4.12, 5.22, 5.23

1. Introduction

The use of formal syntax as the basis of automatically constructed compilers for programming languages has gained wide acceptance in recent years. These applications have, to a great extent, used only the class of context-free languages as a model. For, although features of programming languages such as the requirement that identifiers be declared before use require a more powerful descriptive mechanism, these restrictions can be more efficiently handled by other means (e.g. symbol tables) than by using a more powerful language model. In fact, rather than generalizing the context-free model, practical applications have almost exclusively dealt with more restricted language classes [2]. In particular, the majority of the automatically constructed syntax recognizers are based on a subclass of context-free grammars whose strings can be processed in a single left-to-right no-backup scan, using a pushdown stack. This general approach is formally studied in [3], where the deterministic pushdown automata and the related class of deterministic (context-free) languages are defined. Knuth [4] defines the LR(k) grammars,[1] which have the property that they generate all and only the deterministic languages. (In fact, he shows that the LR(1) grammars also have this "completeness" property with respect to the deterministic languages.) Thus the LR(k) grammars (and, in particular, the LR(1) grammars) are an extremely useful tool for the study of the single-scan context-free languages.

Knuth has described an algorithm for testing an arbitrary context-free grammar for the LR(k) condition (for any particular value of $k \geq 0$). If it succeeds, it generates a deterministic pushdown automaton which can recognize and parse sentences of the given grammar. However, the effort required to make this check and the size of the processor produced grow very rapidly with the complexity of the grammar [1, p. 128]. For a very large grammar, such as one describing a syntax for a programming language, the procedure is not practical. In the first part of this paper a practical method, the basis of which is the Knuth algorithm but which will allow us to work with very large grammars, is investigated. This method involves dividing the given grammar into a number of smaller grammars, proving each part is LR(1) using the Knuth algorithm, and concluding that the original grammar is LR(1) if certain conditions relating the parts are satisfied. The second part of the paper applies this method to a grammar for ALGOL and discusses the implementation of the resulting LR(1) processor.

2. Definitions and Notation

A *vocabulary* V is the union of two disjoint sets of symbols, V_N (the *nonterminal symbols*) and V_T (the *terminal symbols*). For any set of symbols V, the notation V^* is used for the set of all finite-length strings over V, including the empty string, ϵ; $V^+ \triangleq V^* - \{\epsilon\}$. For $k \geq 0$, V^k de-

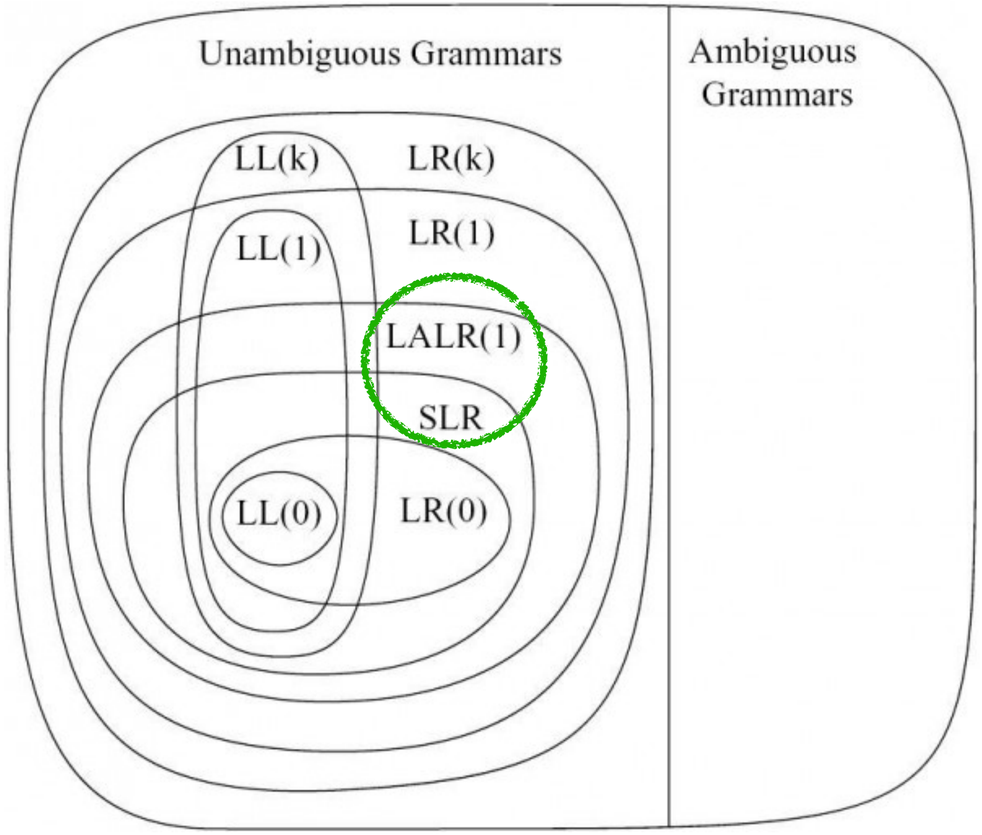
A short version of this paper was presented at the ACM Symposium on the Theory of Computing, Marina del Rey, California, May 5-7, 1969.
* Present Address: Applied Logic Research Institute, Princeton, N. J.

[1] k is a nonnegative integer.

A. J. Korenjak, A practical method for constructing LR(k) processors, Comm. of the ACM, 12(11), 1969

(Intermezzo)

Context-free parsing grammars



Programming Languages — D. Gries, Editor

Simple *LR(k)* Grammars

Franklin L. DeRemer
University of California, Santa Cruz

A class of context-free grammars, called the "Simple *LR(k)*" or *SLR(k)* grammars is defined. This class has been shown to include weak precedence and simple precedence grammars as proper subsets. How to construct parsers for the *SLR(k)* grammars is also shown. These parser-construction techniques are extendible to cover all of the *LR(k)* grammars of Knuth; they have been implemented and by direct comparison proved to be superior to precedence techniques, not only in the range of grammars covered, but also in the size and speed of parser construction and in the size and speed of the resulting parsers.

Key Words and Phrases: context-free grammar, *LR(k)* grammar, precedence grammar, syntactic analysis, parsing algorithm, parser, finite-state machine, deterministic pushdown automaton
CR Categories: 4.12, 5.22, 5.23

Work reported herein was supported in part by Project MAC, an MIT research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

1. Introduction

Some of the results attained in the course of thesis research by the author are reported in this paper. We define a class of context-free grammars, called the "Simple *LR(k)*" or *SLR(k)* grammars, that has been shown [2] to include the weak precedence grammar [9] and the simple precedence grammar [14]. We also show how to construct parsers for *SLR(k)* grammars. The construction technique can be extended [2, 4] to an algorithm for constructing a parser for any *LR(k)* grammar [11]; i.e. any grammar which generates strings each of which can be parsed during a single deterministic scan from left (*L*) to right (*R*) without looking ahead more than *k* symbols.

Our parser-constructing techniques have been implemented [3, 7], and they have been compared with the mixed strategy precedence (MSP) techniques of McKeeman [12]. It was found that in addition to the main advantage of working for a larger class of grammars, our *SLR(k)* techniques are superior to the precedence techniques both in the speed of parser construction and in the size and speed of the parsers produced. Our *SLR*(1) techniques require from one-tenth to one-twentieth the time required by MSP techniques to construct parsers for practical grammars, and the resulting *SLR*(1) parsers require about two-thirds the space and time required for the corresponding MSP parsers.

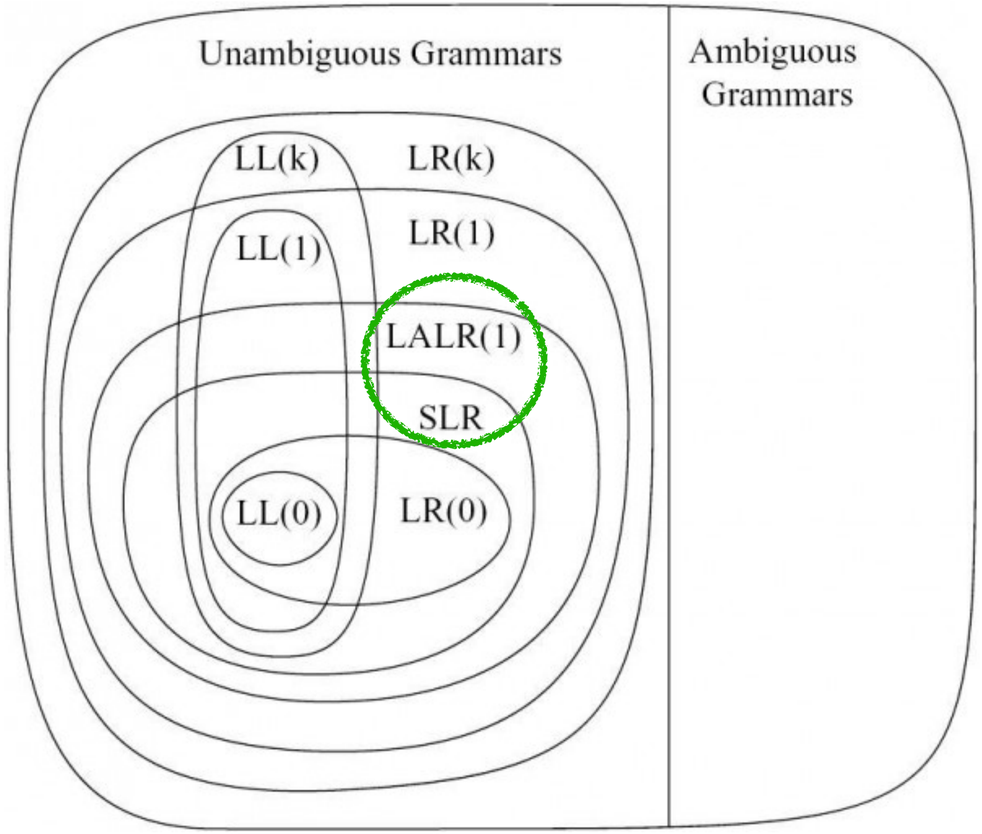
2. Terminology

A *context-free* (CF) *grammar* is a quadruple (V_T, V_N, S, P) where V_T is a finite set of symbols called *terminals*, V_N is a finite set of symbols distinct from those in V_T called *nonterminals*, S is a distinguished member of V_N called the *starting symbol*, and P is a

DeRemer, F. L. (1971). Simple LR(k) grammars. Communications of the ACM, 14(7), 453–460. doi:10.1145/362619.362625

(Intermezzo)

Context-free parsing grammars



- Reduces the number of states one order of magnitude.

Programming Languages — D. Gries, Editor

Simple *LR(k)* Grammars

Franklin L. DeRemer
University of California, Santa Cruz

A class of context-free grammars, called the "Simple *LR(k)*" or *SLR(k)* grammars is defined. This class has been shown to include weak precedence and simple precedence grammars as proper subsets. How to construct parsers for the *SLR(k)* grammars is also shown. These parser-construction techniques are extendible to cover all of the *LR(k)* grammars of Knuth; they have been implemented and by direct comparison proved to be superior to precedence techniques, not only in the range of grammars covered, but also in the size and speed of parser construction and in the size and speed of the resulting parsers.

Key Words and Phrases: context-free grammar, *LR(k)* grammar, precedence grammar, syntactic analysis, parsing algorithm, parser, finite-state machine, deterministic pushdown automaton
CR Categories: 4.12, 5.22, 5.23

Work reported herein was supported in part by Project MAC, an MIT research project sponsored by the Advanced Research Projects Agency, Department of Defense, under Office of Naval Research Contract Nonr-4102(01).

1. Introduction

Some of the results attained in the course of thesis research by the author are reported in this paper. We define a class of context-free grammars, called the "Simple *LR(k)*" or *SLR(k)* grammars, that has been shown [2] to include the weak precedence grammar [9] and the simple precedence grammar [14]. We also show how to construct parsers for *SLR(k)* grammars. The construction technique can be extended [2, 4] to an algorithm for constructing a parser for any *LR(k)* grammar [11]; i.e. any grammar which generates strings each of which can be parsed during a single deterministic scan from left (*L*) to right (*R*) without looking ahead more than *k* symbols.

Our parser-constructing techniques have been implemented [3, 7], and they have been compared with the mixed strategy precedence (MSP) techniques of McKeeman [12]. It was found that in addition to the main advantage of working for a larger class of grammars, our *SLR(k)* techniques are superior to the precedence techniques both in the speed of parser construction and in the size and speed of the parsers produced. Our *SLR*(1) techniques require from one-tenth to one-twentieth the time required by MSP techniques to construct parsers for practical grammars, and the resulting *SLR*(1) parsers require about two-thirds the space and time required for the corresponding MSP parsers.

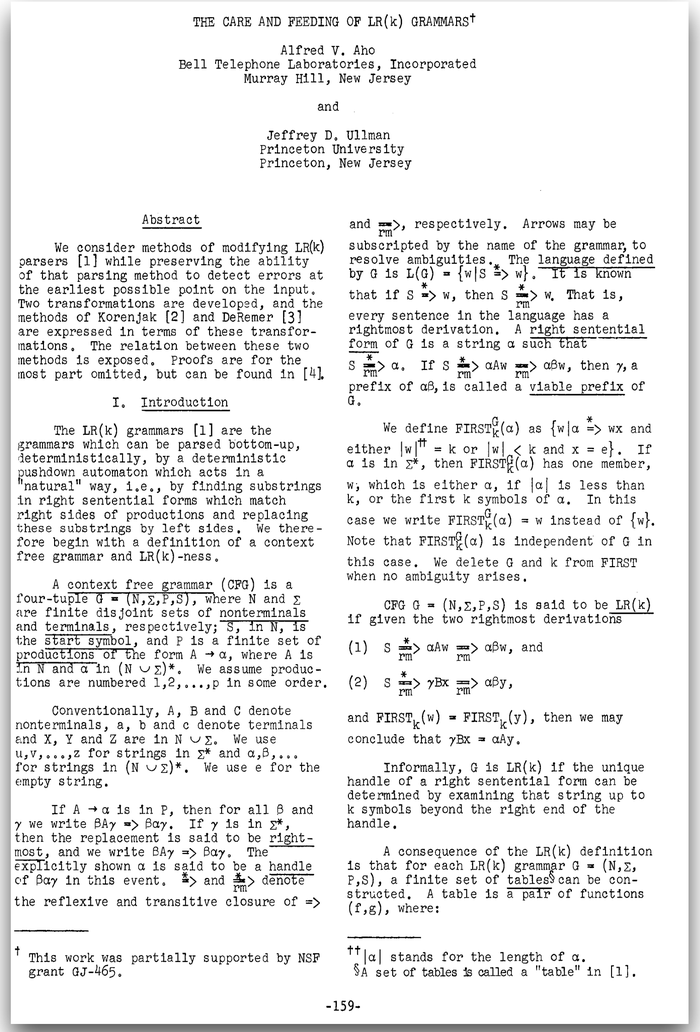
2. Terminology

A *context-free* (CF) *grammar* is a quadruple (V_T, V_N, S, P) where V_T is a finite set of symbols called *terminals*, V_N is a finite set of symbols distinct from those in V_T called *nonterminals*, S is a distinguished member of V_N called the *starting symbol*, and P is a

DeRemer, F. L. (1971). Simple LR(k) grammars. Communications of the ACM, 14(7), 453–460. doi:10.1145/362619.362625

Two papers

The care and feeding of LR(k) grammars



THE CARE AND FEEDING OF LR(k) GRAMMARS†

Alfred V. Aho
Bell Telephone Laboratories, Incorporated
Murray Hill, New Jersey

and

Jeffrey D. Ullman
Princeton University
Princeton, New Jersey

Abstract

We consider methods of modifying LR(k) parsers [1] while preserving the ability of that parsing method to detect errors at the earliest possible point on the input. Two transformations are developed, and the methods of Korenjak [2] and DeRemer [3] are expressed in terms of these transformations. The relation between these two methods is exposed. Proofs are for the most part omitted, but can be found in [4].

I. Introduction

The LR(k) grammars [1] are the grammars which can be parsed bottom-up, deterministically, by a deterministic pushdown automaton which acts in a "natural" way, i.e., by finding substrings in right sentential forms which match right sides of productions and replacing these substrings by left sides. We therefore begin with a definition of a context free grammar and LR(k)-ness.

A context free grammar (CFG) is a four-tuple $G = (N,\Sigma,P,S)$, where N and Σ are finite disjoint sets of nonterminals and terminals, respectively; S, in N, is the start symbol, and P is a finite set of productions of the form $A \to \alpha$, where A is in N and α in $(N \cup \Sigma)^*$. We assume productions are numbered $1,2,\ldots,p$ in some order.

Conventionally, A, B and C denote nonterminals, a, b and c denote terminals and X, Y and Z are in $N \cup \Sigma$. We use $u,v,\ldots,z$ for strings in Σ^* and $\alpha,\beta,\ldots$ for strings in $(N \cup \Sigma)^*$. We use e for the empty string.

If $A \to \alpha$ is in P, then for all β and γ we write $\beta A\gamma \Rightarrow \beta\alpha\gamma$. If γ is in Σ^*, then the replacement is said to be rightmost, and we write $\beta A\gamma \underset{rm}{\Rightarrow} \beta\alpha\gamma$. The explicitly shown α is said to be a handle of $\beta\alpha\gamma$ in this event. $\overset{*}{\Rightarrow}$ and $\overset{*}{\underset{rm}{\Rightarrow}}$ denote the reflexive and transitive closure of $\Rightarrow$ and $\underset{rm}{\Rightarrow}$, respectively. Arrows may be subscripted by the name of the grammar, to resolve ambiguities. The language defined by G is $L(G) = \{w \mid S \overset{*}{\Rightarrow} w\}$. It is known that if $S \overset{*}{\Rightarrow} w$, then $S \overset{*}{\underset{rm}{\Rightarrow}} w$. That is, every sentence in the language has a rightmost derivation. A right sentential form of G is a string α such that $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha$. If $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$, then γ, a prefix of $\alpha\beta$, is called a viable prefix of G.

We define $\text{FIRST}_k^G(\alpha)$ as $\{w \mid \alpha \overset{*}{\Rightarrow} wx$ and either $|w|^{\dagger\dagger} = k$ or $|w| < k$ and $x = e\}$. If α is in Σ^*, then $\text{FIRST}_k^G(\alpha)$ has one member, w, which is either α, if $|\alpha|$ is less than k, or the first k symbols of α. In this case we write $\text{FIRST}_k^G(\alpha) = w$ instead of $\{w\}$. Note that $\text{FIRST}_k^G(\alpha)$ is independent of G in this case. We delete G and k from FIRST when no ambiguity arises.

CFG $G = (N,\Sigma,P,S)$ is said to be LR(k) if given the two rightmost derivations

(1) $S \overset{*}{\underset{rm}{\Rightarrow}} \alpha Aw \underset{rm}{\Rightarrow} \alpha\beta w$, and

(2) $S \overset{*}{\underset{rm}{\Rightarrow}} \gamma Bx \underset{rm}{\Rightarrow} \alpha\beta y$,

and $\text{FIRST}_k(w) = \text{FIRST}_k(y)$, then we may conclude that $\gamma Bx = \alpha Ay$.

Informally, G is LR(k) if the unique handle of a right sentential form can be determined by examining that string up to k symbols beyond the right end of the handle.

A consequence of the LR(k) definition is that for each LR(k) grammar $G = (N,\Sigma,P,S)$, a finite set of tables§ can be constructed. A table is a pair of functions (f,g), where:

† This work was partially supported by NSF grant GJ-465.

†† $|\alpha|$ stands for the length of α.

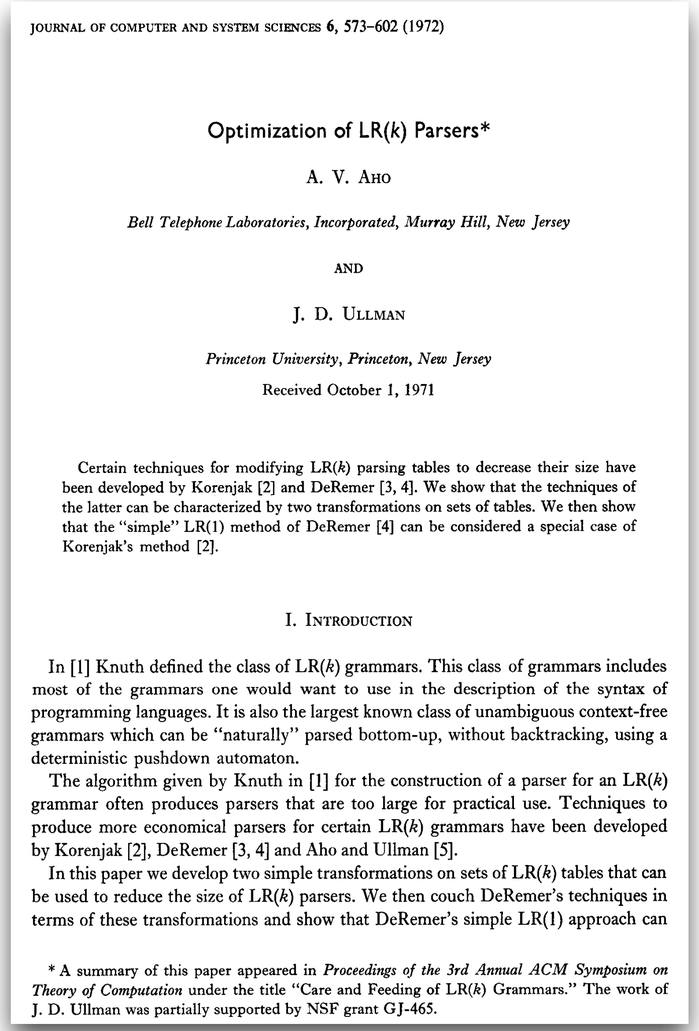
§ A set of tables is called a "table" in [1].

Aho, A. V., & Ullman, J. D. (1971). The care and feeding of LR(k) grammars. Proceedings of the Third Annual ACM Symposium on Theory of Computing - STOC '71. doi:10.1145/800157.805048

Two papers

Optimization of LR(k) Parsers

- They introduce two transformations:
  - Merge of compatible tables.
  - Postponement of error checking.
- Their approach beats both Korenjak's and De Remer's techniques by producing smaller parsing tables.
- They also show that De Remer's approach is a special case of Korenjak's.



JOURNAL OF COMPUTER AND SYSTEM SCIENCES 6, 573–602 (1972)

Optimization of LR(*k*) Parsers*

A. V. AHO

Bell Telephone Laboratories, Incorporated, Murray Hill, New Jersey

AND

J. D. ULLMAN

Princeton University, Princeton, New Jersey

Received October 1, 1971

Certain techniques for modifying LR(*k*) parsing tables to decrease their size have been developed by Korenjak [2] and DeRemer [3, 4]. We show that the techniques of the latter can be characterized by two transformations on sets of tables. We then show that the "simple" LR(1) method of DeRemer [4] can be considered a special case of Korenjak's method [2].

I. INTRODUCTION

In [1] Knuth defined the class of LR(*k*) grammars. This class of grammars includes most of the grammars one would want to use in the description of the syntax of programming languages. It is also the largest known class of unambiguous context-free grammars which can be "naturally" parsed bottom-up, without backtracking, using a deterministic pushdown automaton.

The algorithm given by Knuth in [1] for the construction of a parser for an LR(*k*) grammar often produces parsers that are too large for practical use. Techniques to produce more economical parsers for certain LR(*k*) grammars have been developed by Korenjak [2], DeRemer [3, 4] and Aho and Ullman [5].

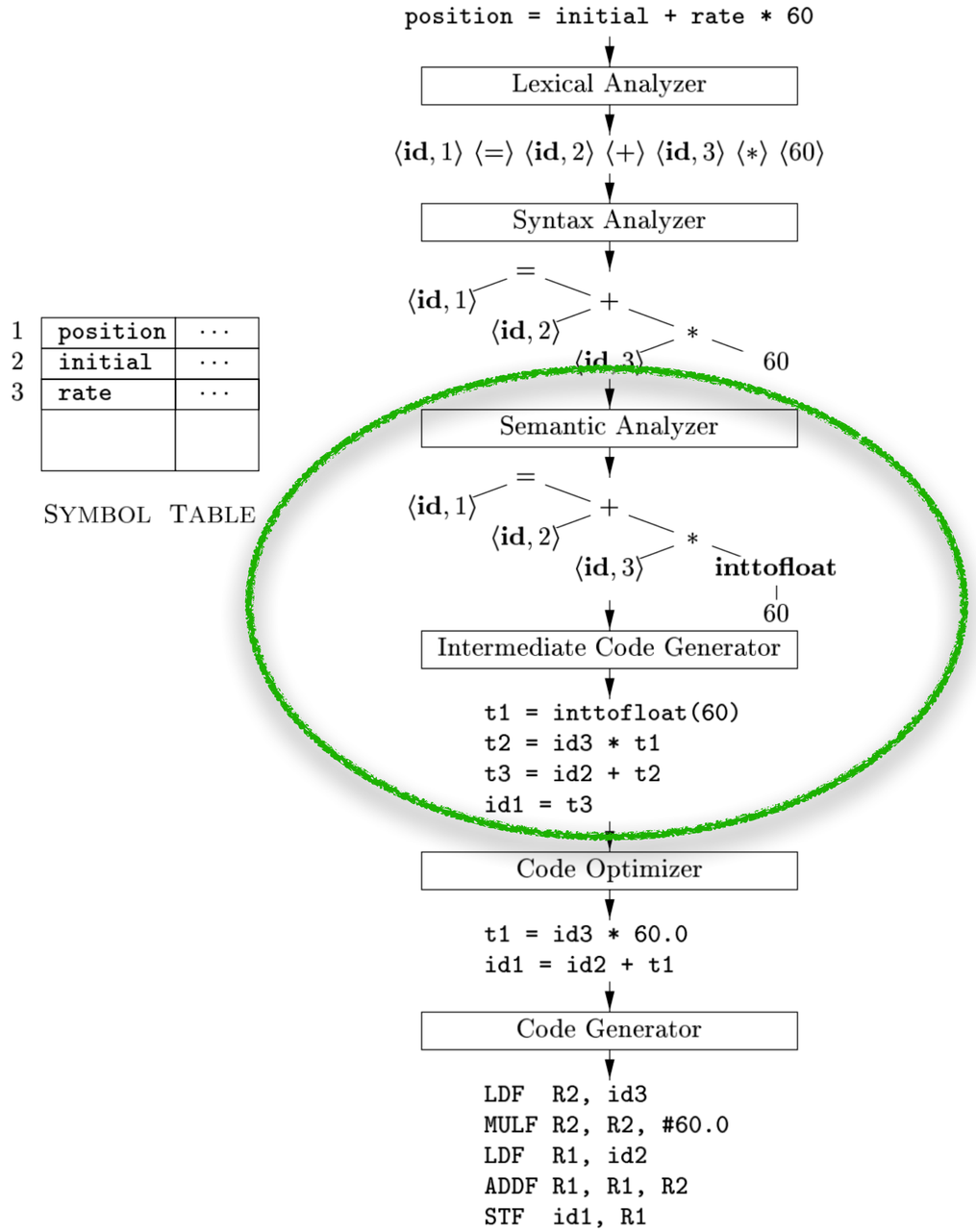
In this paper we develop two simple transformations on sets of LR(*k*) tables that can be used to reduce the size of LR(*k*) parsers. We then couch DeRemer's techniques in terms of these transformations and show that DeRemer's simple LR(1) approach can

* A summary of this paper appeared in *Proceedings of the 3rd Annual ACM Symposium on Theory of Computation* under the title "Care and Feeding of LR(*k*) Grammars." The work of J. D. Ullman was partially supported by NSF grant GJ-465.

Aho, A. V., & Ullman, J. D. (1972). Optimization of LR(k) parsers. Journal of Computer and System Sciences, 6(6), 573–602. doi:10.1016/s0022-0000(72)80031-x

(Intermezzo)

Intermediate-code generation



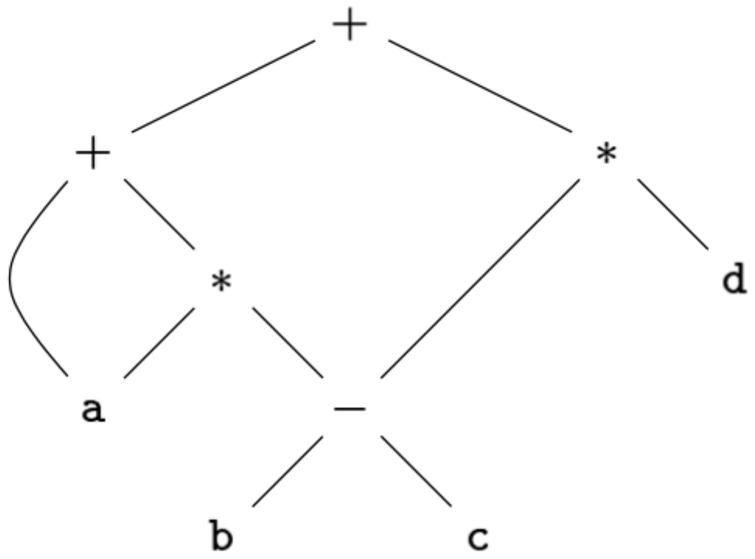
(Intermezzo)

Intermediate-code generation

Arithmetic expression

```
a + a * (b - c) + (b - c) * d
```

Direct acyclic graph



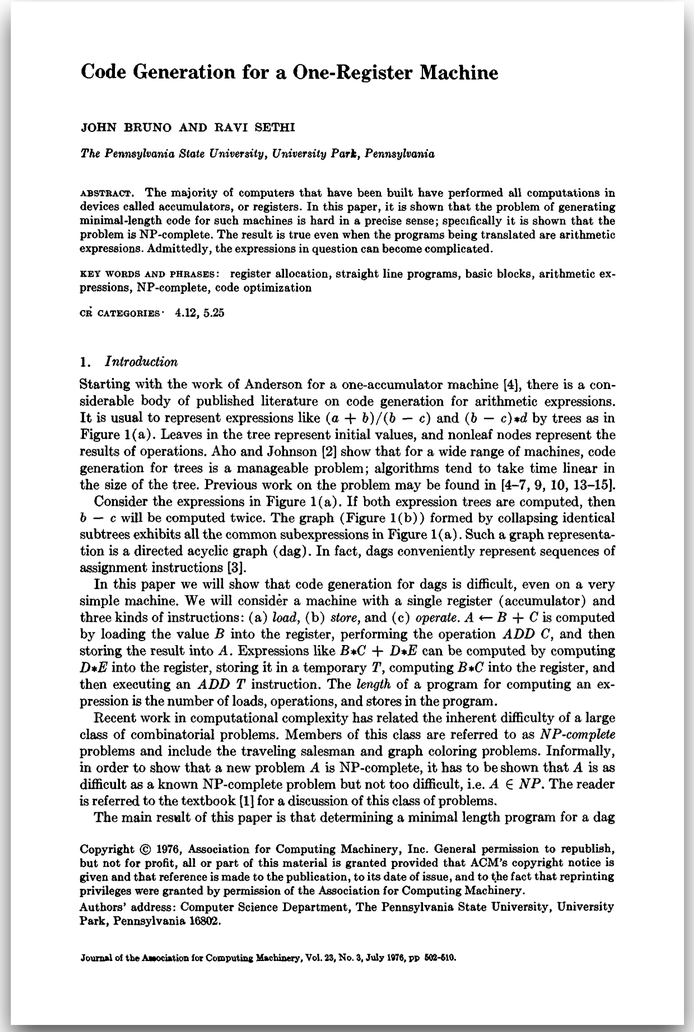
Three-adress code

$$
\begin{aligned}
t_1 &= b - c \\
t_2 &= a * t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 * d \\
t_5 &= t_3 + t_4
\end{aligned}
$$

(Intermezzo)

Intermediate-code generation

- The optimal code generation (OCG) problem is to produce from a DAG the shortest machine program that evaluates and stores all roots of the DAG.
- Bruno and Sethi have shown in this paper that optimal code generation for arithmetic expressions into a one-register machine is NP-complete.
- Their approach transforms 3-SAT into OCG, which produces quite complex DAG.
- They leave open the question of whether or not there could be a more effective way to generate optimal code for simpler classes of DAG.



Code Generation for a One-Register Machine

JOHN BRUNO AND RAVI SETHI

The Pennsylvania State University, University Park, Pennsylvania

ABSTRACT. The majority of computers that have been built have performed all computations in devices called accumulators, or registers. In this paper, it is shown that the problem of generating minimal-length code for such machines is hard in a precise sense; specifically it is shown that the problem is NP-complete. The result is true even when the programs being translated are arithmetic expressions. Admittedly, the expressions in question can become complicated.

KEY WORDS AND PHRASES: register allocation, straight line programs, basic blocks, arithmetic expressions, NP-complete, code optimization

CR CATEGORIES: 4.12, 5.25

1. *Introduction*

Starting with the work of Anderson for a one-accumulator machine [4], there is a considerable body of published literature on code generation for arithmetic expressions. It is usual to represent expressions like $(a + b)/(b - c)$ and $(b - c)*d$ by trees as in Figure 1(a). Leaves in the tree represent initial values, and nonleaf nodes represent the results of operations. Aho and Johnson [2] show that for a wide range of machines, code generation for trees is a manageable problem; algorithms tend to take time linear in the size of the tree. Previous work on the problem may be found in [4–7, 9, 10, 13–15].

Consider the expressions in Figure 1(a). If both expression trees are computed, then $b - c$ will be computed twice. The graph (Figure 1(b)) formed by collapsing identical subtrees exhibits all the common subexpressions in Figure 1(a). Such a graph representation is a directed acyclic graph (dag). In fact, dags conveniently represent sequences of assignment instructions [3].

In this paper we will show that code generation for dags is difficult, even on a very simple machine. We will consider a machine with a single register (accumulator) and three kinds of instructions: (a) *load*, (b) *store*, and (c) *operate*. $A \leftarrow B + C$ is computed by loading the value B into the register, performing the operation *ADD C*, and then storing the result into A. Expressions like $B*C + D*E$ can be computed by computing $D*E$ into the register, storing it in a temporary T, computing $B*C$ into the register, and then executing an *ADD T* instruction. The *length* of a program for computing an expression is the number of loads, operations, and stores in the program.

Recent work in computational complexity has related the inherent difficulty of a large class of combinatorial problems. Members of this class are referred to as *NP-complete* problems and include the traveling salesman and graph coloring problems. Informally, in order to show that a new problem A is NP-complete, it has to be shown that A is as difficult as a known NP-complete problem but not too difficult, i.e. $A \in NP$. The reader is referred to the textbook [1] for a discussion of this class of problems.

The main result of this paper is that determining a minimal length program for a dag

Authors' address: Computer Science Department, The Pennsylvania State University, University Park, Pennsylvania 16802.

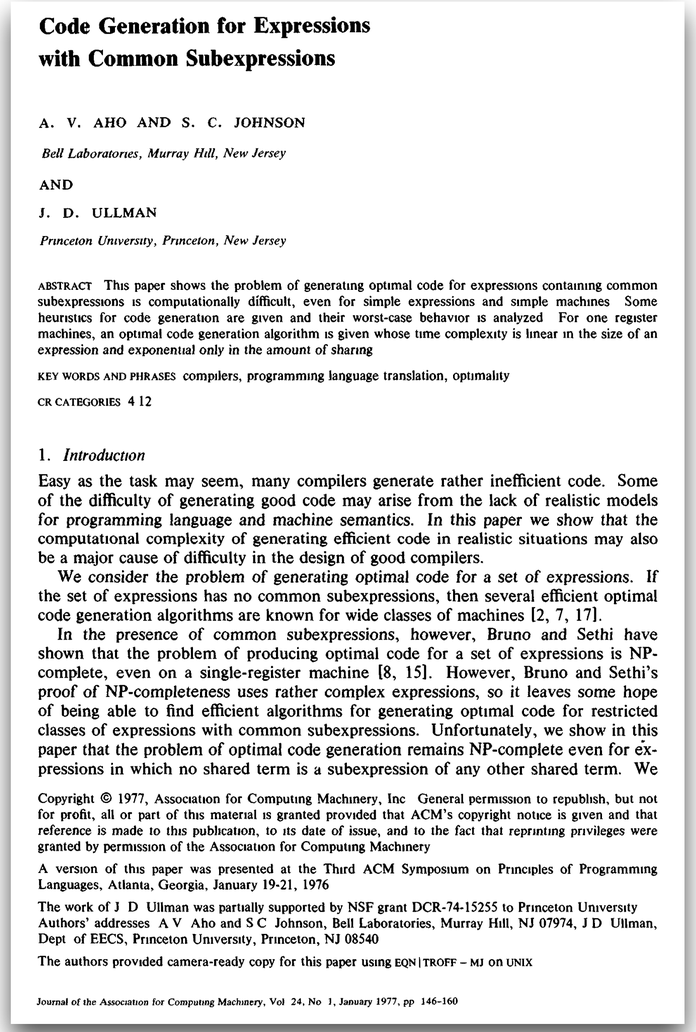
Journal of the Association for Computing Machinery, Vol. 23, No. 3, July 1976, pp 502-510.

Bruno, J., & Sethi, R. (1976). *Code Generation for a One-Register Machine. Journal of the ACM, 23(3), 502–510.* doi:10.1145/321958.321971

Two papers

Code generation for expressions with common subexpressions

- A&U show in this paper that the problem of generating optimal code for arithmetic expressions is NP-complete, even for expressions with no shared subexpressions.
- This is the case even for one-register machines in *level-one* DAG, a DAG where every shared node has leafs as children.



Code Generation for Expressions with Common Subexpressions

A. V. AHO AND S. C. JOHNSON

Bell Laboratories, Murray Hill, New Jersey

AND

J. D. ULLMAN

Princeton University, Princeton, New Jersey

ABSTRACT This paper shows the problem of generating optimal code for expressions containing common subexpressions is computationally difficult, even for simple expressions and simple machines Some heuristics for code generation are given and their worst-case behavior is analyzed For one register machines, an optimal code generation algorithm is given whose time complexity is linear in the size of an expression and exponential only in the amount of sharing

KEY WORDS AND PHRASES compilers, programming language translation, optimality

CR CATEGORIES 4 12

1. *Introduction*

Easy as the task may seem, many compilers generate rather inefficient code. Some of the difficulty of generating good code may arise from the lack of realistic models for programming language and machine semantics. In this paper we show that the computational complexity of generating efficient code in realistic situations may also be a major cause of difficulty in the design of good compilers.

We consider the problem of generating optimal code for a set of expressions. If the set of expressions has no common subexpressions, then several efficient optimal code generation algorithms are known for wide classes of machines [2, 7, 17].

In the presence of common subexpressions, however, Bruno and Sethi have shown that the problem of producing optimal code for a set of expressions is NP-complete, even on a single-register machine [8, 15]. However, Bruno and Sethi's proof of NP-completeness uses rather complex expressions, so it leaves some hope of being able to find efficient algorithms for generating optimal code for restricted classes of expressions with common subexpressions. Unfortunately, we show in this paper that the problem of optimal code generation remains NP-complete even for expressions in which no shared term is a subexpression of any other shared term. We

A version of this paper was presented at the Third ACM Symposium on Principles of Programming Languages, Atlanta, Georgia, January 19-21, 1976

The work of J D Ullman was partially supported by NSF grant DCR-74-15255 to Princeton University

Authors' addresses A V Aho and S C Johnson, Bell Laboratories, Murray Hill, NJ 07974, J D Ullman, Dept of EECS, Princeton University, Princeton, NJ 08540

The authors provided camera-ready copy for this paper using EQN | TROFF – MJ on UNIX

Journal of the Association for Computing Machinery, Vol 24, No 1, January 1977, pp 146–160

Aho, A. V., Johnson, S. C., & Ullman, J. D. (1977). *Code Generation for Expressions with Common Subexpressions. Journal of the ACM, 24(1), 146–160.* doi:10.1145/321992.322001

Two books


The Design and Analysis of Computer Algorithms
AHO | HOPCROFT | ULLMAN
Vertex 1
Vertex 2
Vertex 3 Empty List
Vertex 4
Head Next
Vertices
Edges
Principles of Compiler Design
Alfred V. Aho
Jeffrey D. Ullman

Dragon book

- Alfred V. Aho and Jeffrey D. Ullman. 1977. Principles of Compiler Design (Addison-Wesley series in computer science and information processing). Addison-Wesley Longman Publishing Co., Inc., USA. https://dl.acm.org/doi/book/10.5555/1095594

- Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman. 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley Longman Publishing Co., Inc., USA. https://dl.acm.org/doi/book/10.5555/6448

- Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison-Wesley Longman Publishing Co., Inc., USA, https://dl.acm.org/doi/10.5555/1177220

Describes in ***detail*** all the algorithms for a syntax-directed translation-based compiler, from lexical analysis to code generation.

Significantly extended and improved the optimization techniques chapters

Introduction to Automata Theory, Language and Computation

- Hopcroft, John E.; Ullman, Jeffrey D. (1968). *Formal Languages and Their Relation to Automata*. Addison-Wesley.
- Hopcroft, John E.; Ullman, Jeffrey D. (1979). *Introduction to Automata Theory, Languages, and Computation* (1st ed.). Addison-Wesley. ISBN 81-7808-347-7.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley. ISBN 81-7808-347-7.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2006). *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Addison-Wesley. ISBN 0-321-45536-3.
- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2013). *Introduction to Automata Theory, Languages, and Computation* (3rd ed.). Pearson. ISBN 978-1292039053.

Rajeev Motwani defended his PhD thesis at Berkeley in 1988 on probabilistic analysis of matching and network flow algorithms.

And many more…

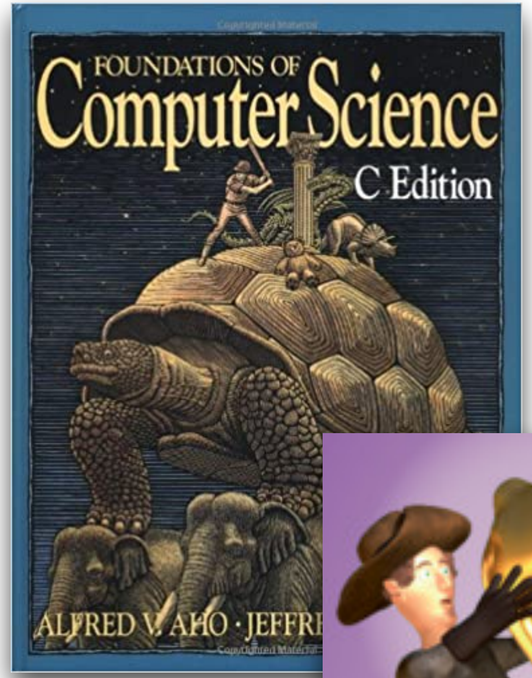
- Ullman's CV lists 19 books until 2006, and many more translations.
- Two freely available web books.

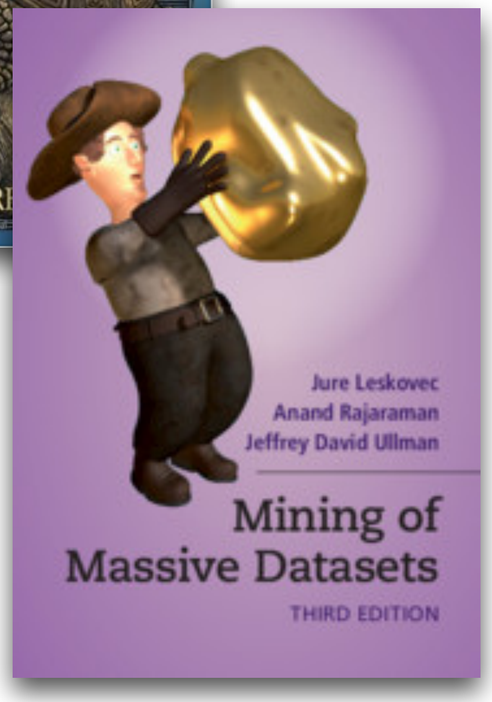
BIBLIOGRAPHY

Books

1. (with J. E. Hopcroft) *Formal Languages and Their Relation to Automata*, Addison–Wesley, Reading, Mass., 1969. Japanese translation, 1971; Czech translation, 1978; unauthorized Chinese translation, 1979.
2. (with A. V. Aho) *The Theory of Parsing, Translation, and Compiling, Vol. I: Parsing*, Prentice–Hall, Englewood Cliffs, N. J., 1972. Japanese translation, 1977; unauthorized Russian translation, 1978.
3. (with A. V. Aho) *The Theory of Parsing, Translation, and Compiling, Vol. II: Compiling*, Prentice–Hall, Englewood Cliffs, N. J., 1973. Japanese translation, 1977; unauthorized Russian translation, 1978.
4. (with A. V. Aho and J. E. Hopcroft) *The Design and Analysis of Computer Algorithms*, Addison–Wesley, Reading, Mass., 1974. Japanese translation, 1977; Russian translation, 1979; Polish translation, 1980; Hungarian translation, 1983.
5. *Fundamental Concepts of Programming Systems*, Addison–Wesley, Reading, Mass., 1976. Japanese translation, 1981.
6. (with A. V. Aho) *Principles of Compiler Design*, Addison–Wesley, Reading, Mass., 1977. Japanese translation, 1979; World student edition, 1982; Spanish translation, 1983.
7. (with J. E. Hopcroft) *Introduction to Automata Theory, Languages, and Computation*, Addison–Wesley, Reading, Mass., 1979. Japanese translation, 1983; Spanish translation, 1983; Slovene translation, 1986; German translation, 1989; Polish translation, 1994;
8. *Principles of Database Systems*, Computer Science Press, New York, N. Y., 1980. Revised edition, 1982. Russian translation of first edition, 1983; unauthorized Chinese translation of first edition, 1985; Japanese translation of second edition, 1985; Indian edition, 1985; Polish translation of second edition, 1988.
9. (with J. E. Hopcroft and A. V. Aho) *Data Structures and Algorithms*, Addison–Wesley, Reading, Mass., 1983. Japanese translation, 1983; Spanish translation, 1985; French translation, 1988; Russian translation, 2000.
10. *Computational Aspects of VLSI*, Computer Science Press, New York, N. Y., 1984. Japanese translation, 1990; Russian translation, 1990.
11. (with A. V. Aho and R. Sethi) *Compiler Design: Principles, Tools, and Techniques*, Addison–Wesley, Reading, Mass., 1986. World student edition, 1987; Japanese translation, 1988; German translation, 1989; Chinese translation, 2001; Russian translation, 2001.
12. *Principles of Database and Knowledge-Base Systems* Computer Science Press, New York, N. Y.: Volume I: *Classical Database Systems*, 1988; Volume II: *The New Technologies*, 1989. Italian translation, 1991.
13. (with A. V. Aho) *Foundations of Computer Science* Computer Science Press, New York, N. Y., 1992. Italian translation, 1993; French translation, 1993; Chinese translation, 1993; Spanish translation, 1994; German translation, 1994; C Edition, 1994.
14. *Elements of ML Programming* Prentice–Hall, Englewood Cliffs, N. J., 1994. Japanese translation, 1996; second edition (ML97), 1998.
15. (with J. Widom) *A First Course in Database Systems* Prentice–Hall, Englewood Cliffs, N. J., 1997, 2002, 2008. Hungarian translation, 1999; Chinese translation 2000; Korean translation 2000; Polish translation 2000; Spanish translation, 2000; Italian translation, 2001; Hungarian translation of the third edition, 2009.

1

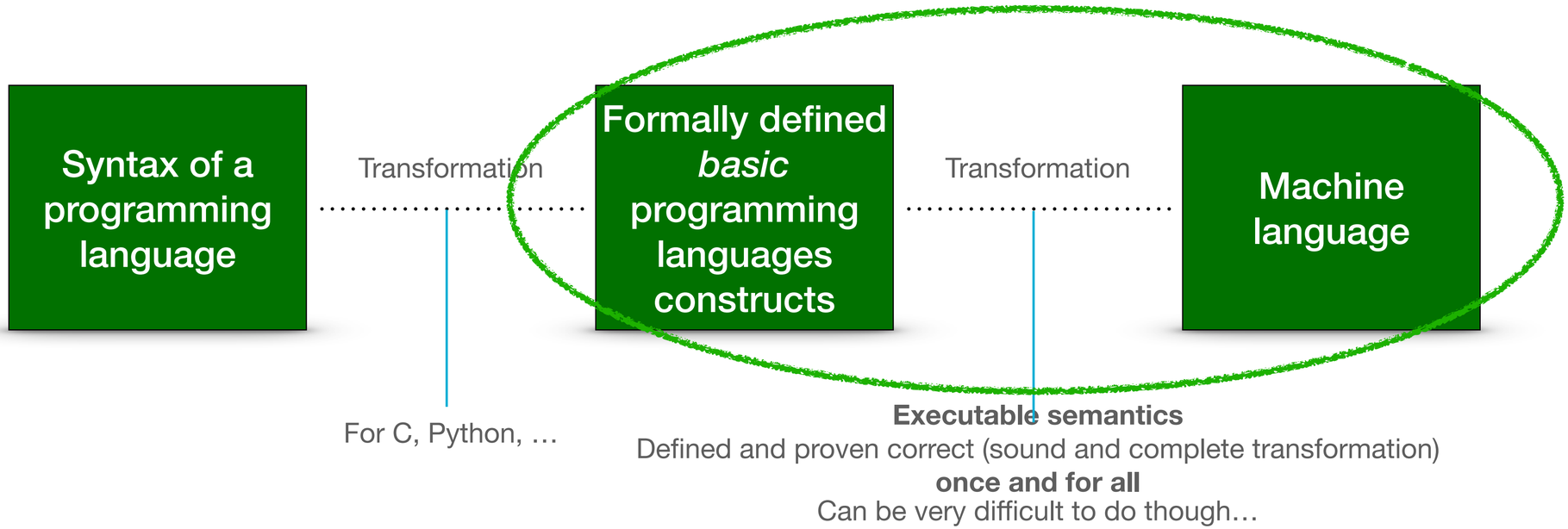




Beyond the dragon

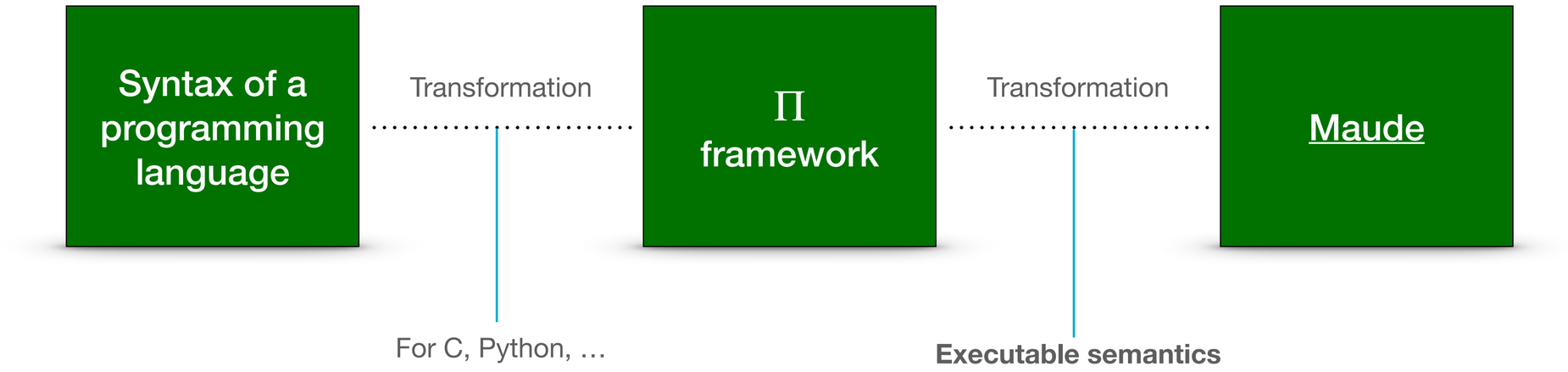
Semantics to the rescue or How to kill the dragon with a light saber?

- A definitive compiler or correct by construction compiler



My 2¢

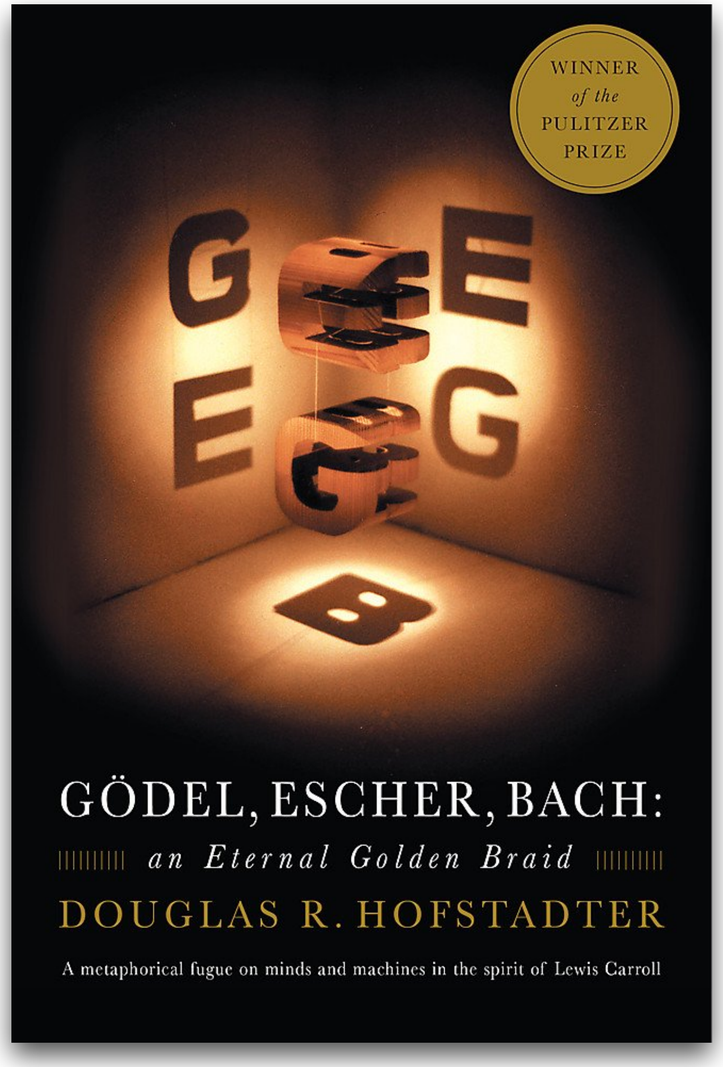
- Correct by construction compiler



Conclusion

- Aho & Ullman's work was and still is indeed a *golden braid*. The fact that they are the recipients of 2020 ACM Turing Award made it also *eternal*.

- This talk is a tribute to their work. But also an acknowledgment for what they have done for Computer Science, and perhaps more importantly, for Computer *Scientists*, myself included, throughout the world.



Thanks!

Aho & Ullman

An Eternal Golden Braid

Christiano Braga - June 23, 2021