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?
ACM Award Honor Laureates

Alfred Vaino Aho is the Lawrence D. Bell Professor Emeritus of Integrated Computer Science and Electrical Engineering/Computer Science from Princeton University. His honors include the IEEE John von Neumann Medal and the NEC Oki Foundation Oki Prize. He is a member of the US National Academy of Sciences, the American Academy of Arts and Sciences, and the Royal Society of Canada. He is a fellow of ACM, IEEE, Bell Labs, and the Americas Association for the Advancement of Science.

Jeffrey David Ullman is the Stanford T. Keck Professor Emeritus at Stanford University and CEO of Databricks, an online learning platform for various computer science topics. He founded the Stanford Research Institute's Information Systems Laboratory (ISL) and co-founded the Harvard Kennedy School’s policy research organization, the Harvard Shorenstein Center on Media and Politics.

2020 ACM A.M. Turing Award Laureates

Alfred Vaino Aho is the Lawrence D. Bell Professor Emeritus of Integrated Computer Science and Electrical Engineering/Computer Science from Princeton University. His honors include the IEEE John von Neumann Medal and the NEC Oki Foundation Oki Prize. He is a member of the US National Academy of Sciences, the American Academy of Arts and Sciences, and the Royal Society of Canada. He is a fellow of ACM, IEEE, Bell Labs, and the Americas Association for the Advancement of Science.

Jeffrey David Ullman is the Stanford T. Keck Professor Emeritus at Stanford University and CEO of Databricks, an online learning platform for various computer science topics. He founded the Stanford Research Institute's Information Systems Laboratory (ISL) and co-founded the Harvard Kennedy School’s policy research organization, the Harvard Shorenstein Center on Media and Politics.
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)
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 Gorrell ("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.

The ACM A.M. Turing Award Laureates 2020

Alfred Vossieck Alho in the Lawrence Radiation Laboratory in Berkeley, California, in 1999. Prior to Columbia, he was the Laboratory Director of Lawrence Livermore National Laboratory where he worked for more than 30 years. A graduate of the University of Toronto, he received his M.S. and Ph.D. in Electrical Engineering Computer Science from the University of California.

Alho’s honors include the IEEE John von Neumann Medal and the ACM [sic] [New] Frontiers in Computing 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, and AAAS, and the American Association for the Advancement of Science.

Jeffrey David Ullman is the Walter E. A. Sherwood Professor Emeritus at Stanford University and is a co-founder of Datanomic, an online-learning platform for various languages. He is also the author of the best-selling "Principles of Database and Knowledge-Base Systems."
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)

2020 ACM A.M. Turing Award Laureates

Alfred Vaino Aho in the Lawrence Guggenheim Professor Emeritus of Computer Science at Stanford University. Aho was born in Riga, Latvia, on August 1, 1934. He received the M.S. degree in electrical engineering from Riga Technological Institute, Riga, Latvia, in 1956. Prior to coming to Canada, he was Vice President of Computing at Xerox PARC in Palo Alto, California, where he worked for more than 30 years. A resident of the University of Toronto, Aho is a member of the Royal Society of Canada, and a fellow of ACM, IEEE, IFIP, and the Canadian Academy of Engineering.

Jeffrey David Ullman is the Walter Isaacson Professor Emeritus of Computer Science at Stanford University and CEO of his company, Informer Technologies, an online learning platform for corporate training courses and data management. He is also the author of numerous books on computer science and programming.
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).

The boys

• Born: 22 November 1942
• Membership in Societies: NAE, NAS, AAAS, ACM, EATCS, SIGACT, SIGMOD, TBPI, SigmaXi.

Excerpt from http://infolab.stanford.edu/~ullman/pub/opb.txt
Foundations of programming language compilers

Background

- Automata theory
- Languages theory
- Compiler principles
Foundations of programming language compilers

Automata theory > Deterministic Finite Automata

\[ \Sigma = \{s_0, s_1, \ldots, s_n\}, \quad L \subseteq \Sigma^* \]
Foundations of programming language compilers

Automata theory > Pushdown Automata

Input tape

Stack

Transition function

Control
Foundations of programming language compilers
Automata theory > Turing machine

Tape

Control

Transition function
Foundations of programming language compilers

Language theory

Chomsky (containment) hierarchy

- Regular languages
- Context-free languages
- Context-sensitive languages
- Recursively enumerable languages
Foundations of programming language compilers

Language theory

Chomsky (containment) hierarchy

- Regular languages
- Context-free languages
- Context-sensitive languages
- Recursively enumerable languages

- Generated by linear grammars
- Accepted by DFA
Chomsky (containment) hierarchy

- Regular languages
- Context-free languages
- Context-sensitive languages
- Recursively enumerable languages

- Generated by context-free grammars
- Accepted by pushdown automata
Foundations of programming language compilers

Language theory > Context-free language grammar

CFG for arithmetic expressions

\[ E \rightarrow E + E \mid E \ast E \mid (E) \mid \text{id} \]

\[ E \Rightarrow E + E \Rightarrow \text{id} + E \Rightarrow \text{id} + E \ast E \Rightarrow \text{id} + \text{id} \ast \text{id} \]

\[ E \Rightarrow E \ast E \Rightarrow E + E \ast E \Rightarrow \text{id} + E \ast E \Rightarrow \text{id} + \text{id} \ast \text{id} \ast \text{id} \]
Foundations of programming language compilers

Language theory > Context-free language grammar

CFG for arithmetic expressions

\[ E \to E + E \mid E \ast E \mid (E) \mid \text{id} \]

CFG for arithmetic expressions with precedence

\[ E \to E + T \mid T \]
\[ T \to T \ast F \mid F \]
\[ F \to (E) \mid \text{id} \]

CFG for arithmetic expressions without left-recursion

\[ E \to T E' \]
\[ E' \to + T E' \mid \epsilon \]
\[ T \to F T' \]
\[ T' \to \ast F T' \mid \epsilon \]
\[ F \to (E) \mid \text{id} \]
Foundations of programming language compilers

Language theory

Chomsky (containment) hierarchy

- Regular languages
- Context-free languages
- Context-sensitive languages
- Recursively enumerable languages

- Generated by context-sensitive grammars
- Accepted by linear-bounded automata
Foundations of programming language compilers

Language theory

Chomsky (containment) hierarchy

- Regular languages
- Context-free languages
- Context-sensitive languages
- Recursively enumerable languages

• 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

\[
A \to \alpha
\]

\[
A \to aA_1A_2 \ldots A_n, \text{ Greibach normal form}
\]

Diagram:

- Transition from state $q_0$ to $q_1$ with input $(\varepsilon, \varepsilon, S)$.
- Transition from $q_1$ to $q_f$ with input $(?, ?, \varepsilon)$.
- Transitions from $q_1$ to itself with inputs $(a_1, A_1, \alpha_1)$, ...
- Transitions from $q_1$ to itself with inputs $(a_n, A_n, \alpha_n)$.
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

Syntax-directed translation
Foundations of programming language compilers

Compiler principles

Syntax-directed translation

Context-free grammar
Foundations of programming language compilers

Compiler principles

Syntax-directed translation

Attribute grammar

```
1 position ...
2 initial ...
3 rate ...

SYMBOL TABLE
```

```
position = initial + rate * 60

Lexical Analyzer

(id, 1) (=) (id, 2) (+) (id, 3) (+) (60)

Syntax Analyzer

(id, 1) = +
(id, 2) +
(id, 3) = 60

Semantic Analyzer

(id, 1) = +
(id, 2) +
(id, 3) = inttofloat(60)

Intermediate Code Generator

t1 = inttofloat(60)
t2 = id3 + t1
t3 = id2 + t2
id1 = t3

Code Optimizer

t1 = id3 + 60.0
id1 = id2 + t1

Code Generator

LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADD R1, R1, R2
STF id1, R1
```
Foundations of programming language compilers

Compiler principles

Syntax-directed translation

<table>
<thead>
<tr>
<th>position</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>initial</td>
<td>...</td>
</tr>
<tr>
<td>rate</td>
<td>...</td>
</tr>
</tbody>
</table>

SYMBOL TABLE

```
Lexical Analyzer
(id, 1) (=) (id, 2) (+) (id, 3) (+) (60)

Syntax Analyzer
(id, 1) = (id, 2) + (id, 3) * (60)

Semantic Analyzer
(id, 1) = (id, 2) + (id, 3) * inttofloat(60)

Intermediate Code Generator
i1 = inttofloat(60)
i2 = id3 * i1
i3 = id2 + i2
id1 = i3

Code Optimizer
i1 = id3 * 60.0
id1 = id2 + i1

Code Generator
LDF R2, id3
MULF R2, R2, #60.0
LDF R1, id2
ADDF R1, R1, R2
STF id1, R1
```
Foundations of programming language compilers

Compiler principles

Syntax-directed translation

<table>
<thead>
<tr>
<th>SYMBOL TABLE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 position</td>
</tr>
<tr>
<td>2 initial</td>
</tr>
<tr>
<td>3 rate</td>
</tr>
</tbody>
</table>

Lexical Analyzer

(id, 1) => (id, 2) (+) (id, 3) (+) (60)

Syntax Analyzer

(id, 1) => (id, 2) (+) (id, 3) => 60

Semantic Analyzer

(id, 1) => (id, 2) (+) (id, 3) => inttofloat(60)

Intermediate Code Generator

t1 = inttofloat(60)
t2 = id3 * t1
t3 = id2 + t2
id1 = t3

Code Optimizer

t1 = id3 * 60.0
id1 = id2 + t1

Code Generator

LDI R2, id3
MULF R2, R2, 60.0
LDI R1, id2
ADDF R1, R1, R2
STF id1, R1

Attribute grammar
Indexed Grammars - An Extension of Context-Free Grammars

Alfred V. Aho

1. Introduction

A language, whether a natural language such as Arabic, is an abstraction and the criteria for the variability of meaning depend on the different assumption toward such a specificity. As a grammar, it is clear that the class of indexed languages is a proper subset of context-sensitive grammars. Moreover, indexed languages resemble context-free grammars and context-sensitive grammars and the class of indexed languages is a proper subset of context-sensitive languages. Hence, there has been a great deal of work in the area of indexed languages. A summary of this paper is presented in this section.

The main result of this paper is that a sentence is a sentence in a language if it is a sentence in an indexed language. The sentence is interpreted through the medium of a stack of indexed languages, called a nested stack automaton, which consists of a finite-state machine with a nested stack as the main auxiliary memory. A program for a nested stack automaton is called a nested stack program, which includes a finite-state machine with a nested stack as the main auxiliary memory. A nested stack program for a nested stack automaton is equivalent to any nested stack automaton which includes a finite-state machine with a nested stack as the main auxiliary memory. Nested stack programs have the computational capability that Turing machine and at each, recursion. The family of nested stack automata is equivalent to the language of nested stack automata which includes a finite-state machine with a nested stack as the main auxiliary memory.
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 that are similar in many respects to those of languages generated by context-free grammars. Moreover, indexed grammars properly include all context-free languages and are 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-structuring language, string specification, context-free grammar, context-sensitive grammar, stack automata

1. Introduction
A language, whether a natural language such as English or a programming language such as APL, in an abstract sense can be considered to be a set of sentences.

Indexed Grammar example:

- **L**: \[ \{a^n b^n c^n \mid n \geq 1\} \]
- **P**: \[ \{S \rightarrow aAfc, A \rightarrow aAgc, A \rightarrow B\} \]
- **F**: \[ \{f = [B \rightarrow b], g = [B \rightarrow bB]\} \]

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 lies strictly between the class of languages generated by stack automata (SA) and the class of context-sensitive languages. Indexed grammars properly include all context-free languages and include 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 grammars, formal language, language theory, automata theory, phrase-structure grammar, phrase-structure language, specification language, context-free grammar, context-sensitive grammar, stack automata

1. Introduction

A language, whether a natural language such as English or a programming language such as APL, in an abstract sense can be considered to be a set of sentences. 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 [15], and various

1. A summary of this paper was presented at the IEEE Eighth Annual Symposium on Switching and Automata Theory, October 1967.
2. The terms “sentence,” “word,” and “program” will be used synonymously as being an element in a language.


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
  
  \[ L_1 = a_1 a_1 a_2 a_3 a_4 L_3 a_5 \]
  
  \[ L_2 = b_1 b_2 \]
  
  \[ L_3 = c_1 L_4 c_2 \]
  
  \[ L_4 = d_1 \]

  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.
On the Capabilities of Codes to Correct Synchronization Errors

JEFFREY D. ULLMAN, MEMBER, IEEE

abstract: Synchronization errors are to be expected when data is transmitted by multi-hop radio links. It is shown, theoreti-
ical and computational results, that codes of certain极具 characteristics can be defined and constructed to correct
such errors.

1. Introduction

A synchronization error is defined to occur whenever a
bit which does not belong appears, or is inserted into a channel
input, where it would not normally be located. The primary
emphasis here will be to consider the capabilities of error
correcting codes, and the effect of such codes on the overall perfor-
mance of a system. The problem, as such, is an extension of
the classical error-correcting problem, in which the basic
idea is that an error in a single bit will not change the
message.

We will consider the case of single synchronization errors
occurring in a channel, and the cases of the error in the
next or the previous bit will be considered. If synchronization errors
will be localized in the body of the code
word. In practice, such a condition is satisfied by cutting a
block of code into fixed-length blocks, among which
are divided by a single synchronization error. The simplest
case is that of a single synchronization error, which
is the case of a single synchronization error.

2. The Types of Synchronization Errors

Let us consider a segment of a code that is to be used as
an error-correcting code. In such a case, the codes
are generally defined as having a minimum distance
between any two code words. This distance is a function of
the number of bits that are different between any two
code words. In the case of a single synchronization error,
the minimum distance is the same as the number of bits
that are different between any two code words.

3. The Consequences of Single Error Correction

The case of a single synchronization error is that of
a single synchronization error, which is the simplest
case. The codes can be divided into two categories:
those which correct synchronization errors, and those
which do not correct synchronization errors. The
primary emphasis here will be to consider the capabilities
of error-correcting codes, and the effect of such codes
on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

4. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
divided into two categories: those which correct
synchronization errors, and those which do not
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

5. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
divided into two categories: those which correct
synchronization errors, and those which do not
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

6. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

7. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
divided into two categories: those which correct
synchronization errors, and those which do not
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

8. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
divided into two categories: those which correct
synchronization errors, and those which do not
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

9. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
divided into two categories: those which correct
synchronization errors, and those which do not
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

10. The Consequences of Multiple Error Correction

The case of multiple synchronization errors is that of
multiple synchronization errors, which is the case
of multiple synchronization errors. The codes can be
divided into two categories: those which correct
synchronization errors, and those which do not
The primary emphasis here will be to consider the
capabilities of error-correcting codes, and the effect of such
codes on the overall performance of a system. The problem,
as such, is an extension of the classical error-correcting
problem, in which the basic idea is that an error in a
single bit will not change the message.

On the Capabilities of Codes to Correct Synchronization Errors

JEFFREY D. ULLMAN, MEMBER, IEEE

Abstract—Two recent papers have dealt with the problem of synchronization errors in the reception of data, and a third paper considered the case of a single word error. We extend and generalize some of the results obtained in these papers, and show the relationships between them.

I. INTRODUCTION

Recent papers have dealt with the problem of synchronization errors in the reception of data, and a third paper considered the case of a single word error. We extend and generalize some of the results obtained in these papers, and show the relationships between them.

II. THE TYPE OF SYNCHRONIZATION ERRORS

Let us consider a stream which contains a sequence of binary symbols, and let a synchronization error occur at some point in the stream. A synchronization error is an error which occurs at the beginning of a block, and which results in the reception of data which is not synchronized with the data transmitted. We may assume without loss of generality that the synchronization error occurs at the beginning of a block, and that the block length is fixed.

III. THE CAPABILITIES OF CODES TO CORRECT SYNCHRONIZATION ERRORS

We now consider the case of a synchronization error occurring in the reception of a block of data. We may assume without loss of generality that the block length is fixed, and that the synchronization error occurs at the beginning of the block.

IV. CONCLUSION

We have shown that the capabilities of codes to correct synchronization errors can be determined by the following conditions:

1. The code must be capable of detecting and correcting all synchronization errors.

2. The code must be capable of detecting any synchronization error which can be corrected by the code.

3. The code must be capable of correcting any synchronization error which can be detected by the code.

REFERENCES


Two papers
(Intermezzo)
Context-free parsing grammars
Context-free parsing grammars

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 to 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(2) 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.

1. 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 ϵ) 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 at 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.

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

- Proposes partitioning the grammar to generate efficient LR(1) parsers.
• Reduces the number of states one order of magnitude.

The care and feeding of LR(k) grammars

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.

Intermediate-code generation
(Intermezzo)
Intermediate-code generation

Arithmetic expression

\[ a + a \times (b - c) + (b - c) \times d \]

Direct acyclic graph

```
          +
         /  \\
        +  *
       /  \\
      a   *  \\
     /     \\
    b     d
```

Three-address code

\[
\begin{align*}
t_1 &= b - c \\
t_2 &= a \times t_1 \\
t_3 &= a + t_2 \\
t_4 &= t_1 \times d \\
t_5 &= t_3 + t_4
\end{align*}
\]
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.

(Intermezzo)

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 leaves as children.

Two books
Dragon book


Significantly extended and improved the optimization techniques chapters.

Describes in detail all the algorithms for a syntax-directed translation-based compiler, from lexical analysis to code generation.
• Hopcroft, John E.; Ullman, Jeffrey D. (1968). *Formal Languages and Their Relation to Automata*. Addison-Wesley.

And many more...

- Ullman’s CV lists 19 books until 2006, and many more translations.

- Two freely available web books.
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

Syntax of a programming language

Transformation

For C, Python, …

Formally defined basic programming languages constructs

Transformation

Executable semantics
Defined and proven correct (sound and complete transformation)

once and for all
Can be very difficult to do though…

Machine language
• 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