Technical Reports Mathematics/Computer Science
FB IV - Department of Computer Science
University of Trier
54296 Trier

Theorietag 2017

Automaten und Formale Sprachen

(Bonn, 18. September – 22. September 2017)

Herausgeber: Henning Fernau

Technical Report No. 17-2 September 2017

Vorwort

Theorietage haben eine lange Tradition bei verschiedenen Fachgruppen der Gesellschaft für Informatik (GI).Die Fachgruppe Automaten und Formale Sprachen (AFS) aus dem Fachbereich Grundlagen der Informatik (GInf) innerhalb der GI trifft sich einmal jährlich zu ihrem zweitägigen Theorietag. In diesem Jahr feiert die Reihe ihre 27. Auflage in Bonn.

Allerdings wird das Thema *Lehre* nur selten thematisiert, obwohl beispielsweise die Mitgestaltung bei Vorschlägen für curriculare Standards der Informatikstudiengänge seitens der Gesellschaft für Informatik zu den Aufgabenbereichen der Untergliederungen der GI gehört. Nach fast zwanzig Jahren war es daher wieder an der Zeit, sich diesem Thema in einem besonderen Workshop zu widmen.

Diese curriculare Diskussion sollte auch in einem internationalen Rahmen erfolgen, weshalb die folgenden Zeilen in englischer Sprache abgefasst sind, ebenso wie die meisten Beiträge. Deshalb ist auch dieses Vorwort nachfolgend englischsprachig.

Formal Languages in Today's and in Future Curricula of Computer Science Programs

Formal language courses have become an integral part of Computer Science (CS) curricula all over the world. Although many things have been changing over time in CS, the material that is currently taught in such classes is mostly relying on results that were established fifty years ago. Even the textbooks usually recommended for such classes have not changed that much over the past decades. This is quite surprising in the quickly changing world of CS.

The contributions to this report touch quite different aspects of teaching Formal Languages.

- How does this subject fit into CS-related curricula like *Software Engineering*?
- Could teaching Formal Languages be split and shared if quite different groups of students attend the courses?
- Is the use of Turing machines as the core model of computability still adequate?
- How can the teaching of Formal Languages be integrated into a curriculum with less focus on theoretical foundations? Programming with Python could be a starting point.

- How is the situation in Finland and Romania, two of the (historically speaking) core countries of Formal Language Theory?
- What about India, a country with quite some tradition in Formal Languages?
- How is the situation in a country like Nigeria, where only very few experts of Formal Languages could carry the torch of this discipline for future generations?
- What is the impact and what are the requirements of Formal Languages outside academia?

Also, the background of the audience was quite diverse. This is true both for the geographical variation, as people from all over the world were attending our meeting, and for the variation in types of universities or audiences for which courses in Formal Languages had to be designed. Also, some people from outside academia attended this workshop and contributed by sharing their views.

We believe that this meeting was a good starting point. This type of meeting should not see another two decades pass before repeating this type of discussion. Rather, it would be good to repeat this experience every 3-5 years.

Henning Fernau

im Namen der gesamten FG-Leitung
on behalf of the whole Board of the Special Interest Groups Automata and Formal Languages

Appendix

In order to give an impression of this workshop, and also since not all talks have some reflection in the following articles, we attach the program below. This also shows the diversity of the talks and should be compared to the Table of Contents of the collection of abstracts that follows.

Program on Thursday

1. Session: national experiences

- Henning Fernau: Short Opening
- Jürgen Dassow: Experiences from German Universities I
- Klaus-Jörn Lange: Experiences from German Universities II
- Daniel Neider: Experiences from German Universities III
- Peter Leupold: Formal Languages and Automata on stackoverflow

2. Session: applied perspectives

- Anna Kasprzik: Formal Languages in Libraries
- Stefan Gulan: Formal Methods in Industry
- Heinz Faßbender: Situation at HS Aachen
- Heinz Schmitz: Situation at HS Trier
- Martin Müller: Theory in Practice

3. Session: an international view

- D. G. Thomas: Formal Languages in the B.Sc. and M.Sc. in India
- Rufus Oladele: Formal Languages in Nigeria
- Mikhail Volkov: Formal Languages in Russia
- Rudolf Freund: What about Austria?

4. Session: concrete suggestions

- Markus L. Schmid: Compiler Construction and Parsing
- Henning Bordihn: Formal Languages in Software Engineering
- Klaus Reinhardt: Advocating abandoning Turing

First Impressions Collected

Program on Friday

1. Session: national experiences

- Henning Fernau: Short Opening
- Joel Day / Mitja Kulczynski: Experiences from the North
- Philipp Kuinke: Formal Language Teaching at RWTH Aachen

2. Session: Needs for now / for the future

Contents

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 1–3.

Formal Languages in Software Engineering

Henning Bordihn

Institut für Informatik, Universität Potsdam
August-Bebel-Straße 89, 14482 Potsdam, Germany
henning@cs.uni-potsdam.de

Abstract

This paper discusses some content of Theoretical Computer Science that is needed in more applied courses such as Software Engineering or in practice. It aims to demonstrate that several aspects of traditional courses about Theoretical Computer Science form the fundament on which applied disciplines build their methods. As an example, the modern paradigm of model driven software development is considered in more detail.

1. Introduction

The author is experienced not only in teaching Theoretical Computer Science, but mainly as a lecturer in courses about Programming, Algorithms and Data Structures and Software Engineering. It turned out that knowledge and methods that are typically taught in Theoretical Computer Science courses are needed in certain parts of other, more application oriented lectures. In order to provide students with necessary prerequisites, Theoretical Computer Science courses should impart knowledge about "ways of thinking" such as modeling, abstraction, precision, and so on. The content used to convey such qualification is, in principle, exchangeable. But one cannot deny that there is some "core" of traditional contents of Theoretical Computer Science that should be pinned to a fundamental course in that field, because it is immediately needed in other courses or in practice. In Software Engineering, for instance, several formal or semi-formal methods require adequate knowledge about theoretical foundations. For example, *model checking* requires knowledge about both state machine models and (temporal) logic, system analysis about learning (of state machine models) from positive data, etc.

This paper aims to identify some major part of such "core" with the help of the course *Compiler and Program Transformation* that I regularly teach at the University of Potsdam. This course considers many aspects of traditional compiler construction, mainly from the phases of lexical, syntactic and semantic analysis, with the goal of applying them in a project dedicated to the paradigm of model driven software development. Several Bachelor's theses that I've supervised in cooperation with partners from industries emphasize the practical relevance of the topic, see, e.g., [1, 3].

This contribution refers to a course at the University of Potsdam that is based on a course co-developed by Bernhard Steffen and Oliver Rüthing, TU Dortmund.

2. Compiler and Model Driven Software Development

In model driven software development (MDSD), models of the system under development form the development artifacts. From the models, code in some target language that is ready for compilation is automatically generated, see, e.g., [2]. There is a lot of benefit:

1. Verification is easier at the more abstract model level, where, for instance, model checking can be used. The automatic code generation guarantees that properties verified at the model level are preserved.
2. Manual coding errors and repetitive implementation work can be avoided.
3. The target platform can easily be exchanged, while the models with their properties are re-used.

The models can be viewed as code in some domain specific, modeling language. In many cases, such domain specific languages (DSL) are specified in extended Backus Naur Form (EBNF). This specification is frequently done by those developers that strive for applying MDSD in projects, but when no adequate DSL is at hand. This part of the work requires all the knowledge about language design, including

- regular expressions for the specification of tokens,
- context-free grammars and languages, in particular LL or LR grammars for the syntax specification,
- limits of context-free models for the decision on which aspects have to be moved to the definition of the static semantics of the DSL.

Finally, a code generator (in most cases template based) must be implemented, completing the translator, that is, the compiler needed for code transformation into some target language.

Nowadays, there exist several tools integrating all these aspects of language infrastructure (scanner and parser generators, platforms for defining and checking static semantic constraints, code generators). But those tools can be used successfully only if the theoretical concepts are well understood.

Therefore, in the course *Compiler and Program Transformation*, the concepts from both compiler construction and MDSD are discussed. These items strongly depend on the corresponding theoretical concepts. Then, in a project, the students develop a DSL called *Mini Game Programming Language* (MGPL) that is suitable for the definition of simple Arcade games. As test cases, specifications of the games Pong and Space Invaders in MGPL are handed to the students. The project goal is to compile these MGPL programs into Java code. If successful, the generated Java code is generated "at the push of a button" and can be compiled and executed by means of the JDK for Java SE, where not a single line Java code must be written by hand.

3. Conclusion

The experience with teaching applied computer science courses at the university level led to some insight of what knowledge should (at least) be gained from fundamental courses about

Theoretical Computer Science. This contains regular expressions and finite automata, context-free grammars and pushdown automata, limits of regular and context-free languages (pumping lemmata), computability and undecidability, basics of complexity theory (e.g., the classes **P** and **NP**, reductions and completeness). The discussion of some properties of the respective models is, to some extend, also desirable.

Next to those topics related to automata and formal languages, decent knowledge about logic (at least propositional and first order logics) is needed[1]. One might wish to consider the interrelations of subjects like formal languages on the one hand and logic on the other, in fundamental theory courses, but one should be aware that the basic concepts need a consideration for themselves first. Looking at the way how the concepts are applied, for instance, in Software Engineering, then a traditional introduction of those concepts appears to be indispensable. Given a curriculum such as the one in Potsdam, the "core" concepts can be supplemented by more advanced concepts, properties or connections to other terms.

References

[1] D. HERGASS, *Evolution eines Metamodells einer Sprache für die Modellierung von Automotive Software*. Bachelor thesis, University of Potsdam, 2017.

[2] S. KELLYL, J. TOLVANEN, *Domain-Specific Modeling: Enabling Full Code Generation*. John Wiley & Sons, 2008.

[3] F. SOREK, *Migration eines Editors zur Modellierung von Automotive Software von Graphiti zu Sirius*. Bachelor thesis, University of Potsdam, 2017.

[1] At the University of Potsdam, there is a separate course about logic for computer scientists. Therefore, this article focuses on automata, formal languages, computability and complexity.

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 4–6.

Modes of Thought from Theoretical Computer Science in Library-Related Development and Research

Anna Kasprzik

Technische Informationsbibliothek
Welfengarten 1B
30167 Hannover
anna.kasprzik@tib.eu

Abstract

This is a personal account of a career in the library sector after an education in formal linguistics and theoretical computer science. The author would like to argue that without her formal training the pursuit of a career in library-related development would not have been possible, and strongly advises against the reduction of such topics in today's curricula.

1. A career in library-related development and research

The author has pursued and obtained an M.A. degree in computational linguistics, theoretical computer science, and cognitive psychology, and during her studies has focussed on formal (tree) languages, formal semantics, and predicate logic. She went on to receive a Ph.D. degree in theoretical computer science, doing research on "formal tree languages and their algorithmic learnability". The key component in all those activities was the concept of "structure", and the question how structures can be generated, recognized, transformed, and validated.

After her Ph.D. the author decided to obtain an additional qualification as a scientific subject librarian since she expected to find interesting aspects from information science in that area and hoped to be able to combine them with structural considerations. The corresponding two-year course addresses candidates with a Ph.D. or a M.A. degree and consists of a one-year practical internship at a scientific library, and a one-year course of theoretical training. The author particularly focussed on topics in the cross-section of the library as a provider of modern digital services and information technology, i.e., within the realm of knowledge engineering, such as controlled vocabularies, semi-automated classification and indexing, text and data mining, semantic query enrichment, modelling of library metadata for various purposes.

During her practical internship at the University Library of Konstanz the author had the opportunity to design and implement an algorithm that translated the in-house subject classification into a tree structure with the aim to eliminate elements that did not comply with the principles of hierarchization implicitly contained in the classification. The author feels that she did benefit considerably from her formal training in accomplishing this task, especially formal tree language theory and complexity (a transition from searching the entire tree to merely

searching the path to the root at one point shaved off two hours of runtime when applying the algorithm to the entire database), and also some modeling principles from software engineering.

After having worked in an IT project with the goal to establish a cloud-based infrastructure for library metadata and a mapping between two metadata formats for one and a half years, the author felt that the particular environment she was working in functioned mainly based on pragmatic considerations and did not set its priorities on conceptual thinking and formal specification. She decided to apply for a position at a library (TIB) that as a member of the Leibniz Association deliberately affords its own department for research and development, and started to work as an expert on domain-specific vocabularies, ontologies, and semantic technologies. Among her tasks was the structural optimization of a thesaurus that had been compiled from several others and consequently had to be complemented with a top-level hierarchy in order to create a homogeneous progression from general to more specialized topics. Again, her previous engagements with formal tree structures and predicate logic provided the author with the foundations she needed in order to work with the underlying semantic network.

In July 2017, TIB (which acts as the national library for technical subjects) has recruited a new Head of Library with the additional requirement to assume the professorship for Digital Libraries and Data Science at the department of Computer Science. Consequently, the library's department for research and development is now being extended considerably, and the author has obtained a position as a junior research group leader. One focus of the research group will be on topics from the industrial realm of the "Smart Factory" (semantic layers for industrial production processes) and the other on library-related knowledge organization systems.

2. Lessons learned

Thus, the components in her curriculum from which the author has stood to benefit most were: Formal (tree) languages including tree structures with additional relations between nodes, some complexity theory, modeling principles from software engineering, logic and formal semantics. Although the author had not acquired extensive programming skills during her studies (only some basic training in C++ and Prolog in order to understand the foundations of programming), her background in theoretical computer science has enabled her to master the main features of a programming language reasonably quickly in order to implement the functionalities needed for her tasks, from Ruby code to small scripts in Perl and Python, to visualizations based on JSON and D3. Most of those tasks were aiming at the manipulation of (hierarchical) tree structures. Additional topics in her working environment (not necessarily in her personal task portfolio, and the following list is not exhaustive) include document classification via vectorization and subsequent formal concept analysis based on lattice theory, the extraction of machine-readable semantic queries from text via natural language processing methods, ontology design based on formal logic as a foundation for non-trivial question answering systems, and a range of topics from the area of machine learning (including deep learning with neural networks).

It is fair to say that libraries are only just beginning to see the benefits of employing computer scientists with a theoretical background. As of 2017, most job offers describe activities in the IT department (administration, hosting, support) and at the lower end of the German civil service pay scale (not suited to attract academically trained computer scientists), rather than activitites in research and development involving conceptualization and hiqh-quality specification

and documentation. This is a lost opportunity and is all the more unfavourable since at a closer look librarians and theoretical computer scientists are very much alike in their semantic way of thinking and building structures in order to organize data and to model the domains in question. The methods needed to do so are all core topics of theoretical computer science.

What is missing is a general education among decision makers in libraries what theoretical computer science is and what it can do for them – and who better than the "Gesellschaft für Informatik" to raise this kind of awareness! Decision makers should be encouraged to create more jobs in their IT and research and development departments that deal with conceptualization, classification, standardization, and handling of metadata on a higher level. Computer scienctists can function as intermediaries between library and IT, thus counteracting the pressure of sacrificing formal standards and conceptions to pragmatic aspects of the modern working environment and making a major contribution towards the development of more sustainable services, to everybody's benefit. An obvious precondition of such a fruitful symbiosis is of course that in the decades to come academically trained computer scientists still *exist*, particularly those with a training in classical theoretical topics such as formal language theory, computability and complexity, logics, and various methods of modeling, and a shift in curricula towards a more short-sighted, pragmatical approach would be detrimental to say the least.

Consequently, not only should curricula in computer science at universities contain the theoretical topics listed above as a (compulsory) standard, but topics such as formal language theory, logics and some elements of formal linguistics should be taught in German secondary schools as well as a general preparation towards a structural and scientific mode of thought.

References

[1] A. KASPRZIK, *Formal tree languages and their algorithmic learnability*. Dissertation, Universität Trier, 2012. Available online at: `http://ubt.opus.hbz-nrw.de/volltexte/2012/737/`.

[2] A. KASPRZIK, *One-shot learning and the polynomial characterizability barrier*. Technical Report 12-4, Universität Trier, 2012. Available online at: `http://www.informatik.uni-trier.de/~kasprzik/oneshotpol.pdf`.

[3] A. KASPRZIK, *Projektbericht : Implementierung eines Hierarchisierungsalgorithmus' für die Konstanzer Systematik*. Technical report, Bibliothek der Universität Konstanz, 2013. Available online at: `https://kops.uni-konstanz.de/handle/123456789/24166`.

[4] A. KASPRZIK, Automatisierte und semiautomatisierte Klassifizierung – eine Analyse aktueller Projekte. *Perspektive Bibliothek* **3** (2014) 1, 85–110.

[5] A. KASPRZIK, Vorläufer internationaler Katalogisierungsprinzipien. *Perspektive Bibliothek* **3** (2014) 2, 120–143.

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 7–9.

Experiences from the North

Mitja Kulczynski

Kiel University, Department of Computer Science,
mku@informatik.uni-kiel.de

Abstract

Following the demand of different study programs and for pedagogical reasons we have changed the way we teach theoretical foundations of computer science in Kiel since the Winter term 2016. The course was split into two parts: An informal and formal round. These parts were lectured consecutively. This report describes our experiences with this approach.

1. Introduction

Teaching theoretical foundations of computer science is never an easy task. At the Kiel University this course is one of the obligatory modules for undergraduate bachelor students. The course starts with teaching formal languages. Students are introduced to regular, context free, context sensitive and unbounded languages. We are teaching the basic automata models and properties of grammars for each category. Afterwards we tackle computability of languages. We cover decidability aspects of previously seen languages and teach mechanisms of how to prove or disprove it. The last part of this course covers basic knowledge about complexity theory. We cover some of the most common time and space complexity classes. Again we teach how to prove which complexity class the previously seen languages belong to.

The lecture is based on a variety of books [1, 2, 3, 4, 5, 6, 7, 8]. We used these books to write our own lecture notes which are provided to every student. In each term this course is given in two lectures and one tutorial with supporting exercises, per week. Therefore the course consists of 26 lectures and about 13 exercise sessions.

Since 2016 we had to face an even harder task at Kiel University: Not only computer science students, but also business informatics students had to participate in this course. The major difficulty was not the lack of knowledge, but the circumstances: The course was designed to achieve 8 credit points, whereas business informatics should receive only half of the points.

In this report we briefly describe how we reached this goal and what the undergraduates thought of our decisions. The last section also covers some thoughts for the future which were inspired by the discussions after the talk at the *Theorietag 2017* in Bonn. All given information is based on having taught the course in this form twice so far.

2. The course

To address the constraint mentioned above we decided to divide the lecture into two parts: A more or less "informal" part, which only covers definitions, examples, and some algorithms from the topics introduced in *Section 1* and a "formal" part where we prove the correctness of previously given constructions and enrich the topics with enhanced background knowledge. The associated tutorials and their exercise sheets follow the same style.

2.1. The lecture

We decided to cover the most common topics related to formal languages. The task was to give a smaller lecture in half of the normally given time without loosing relevant information. Therefore we decided to hide technical details and correctness proofs in the first part of the lecture. Proving the correctness of all covered topics usually takes most of the time. This allows us to cover even more language classes than before e.g. context sensitive languages which were not covered in the previous structure of this course. To fulfil the needs of a business informatics graduate it is sufficient to have an intuition about formal languages, computability and complexity without the complete background knowledge. Also the undergraduates in computer science benefited from this structure. Due to the circular procedure of the course as given in *Figure 1*, they face all topics twice. Based on their previously gained knowledge they are able to focus on the construction proofs without thinking too much about the basic definitions. A problem which appeared in the first run of the course however, was the absence of easy topics in the second half. So our solution was to move some algorithms from the first half into the second half. This arbitration relaxed the difficulty. Unfortunately the formal part of the course throw some students off track.

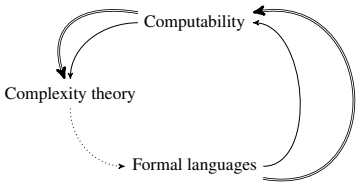


Figure 1: Structure of the course

2.2. The tutorial

Each week the students are able to do homework while supported by an employee. The undergraduates discuss their problems in small groups of 2 to 4 people. The main advantage of this tutorial style is the personal assistance by a tutor based on their current knowledge level. Due to the small group size the tutorial lacks open discussions which are often helpful to tackle general questions. To remove this minus we will add an additional tutorial where an employee solves exercises with a larger audience and guides open discussions to deepen the undergraduates' knowledge.

2.3. Impressions

Our and also the students' reaction was mostly positive. Within the informal part of the course there was a high willingness to participate. This is not too common in theoretical courses. During the tutorials we usually asked the students for their own opinion. The main negative

aspects from their point of view were the lack of difficult topics and the relevance in general. Another negative aspect was the coarse covering of the decidability part. We saw no other way of covering this topic without introducing proofs.

The second run offered us some great opportunities: We were able to directly adjust the level of detail for all topics based on the students' needs. Unfortunately some students were still not able to follow this run.

3. An Outlook

The next run of this course will take place in summer 2018. We will keep up our tutorial style but add an extra tutorial in the classical sense where we will solve additional exercises. To give the students an even better understanding of our topics we consider extending the exercises with practical ones e.g. implementing a pushdown automaton in Python.

The lecture will probably cover fewer topics in the next run. We will try to get rid of the too coarse topics by adding even more examples. This will remove the rough feeling of the decidability and complexity part in the first half of this course.

Some students complained about the circular structure of the lecture. Therefore a recommendation at the *Theorietag* 2017 was to cover one topic each week, where the first lecture covers the informal part and the second one the proofs. We will not attempt this approach since we will lose one of the biggest benefits from our current course style: Revisiting all topics twice and adjusting the lecture to the students' needs.

Overall we just have to make some small adjustments. This is also based on the supporting feedback during the *Theorietag* 2017.

References

[1] J. E. HOPCROFT, R. MOTWANI, J. D. ULLMAN, Automata theory, languages, and computation. *International Edition* **24** (2006).

[2] D. C. KOZEN, *Theory of computation*. Springer Science & Business Media, 2006.

[3] D. C. KOZEN, *Automata and computability*. Springer Science & Business Media, 2012.

[4] A. MAHESHWARI, M. SMID, Introduction to Theory of Computation. *Free Online* (2012).

[5] J. C. MARTIN, *Introduction to Languages and the Theory of Computation*. 4, McGraw-Hill NY, 1991.

[6] U. SCHÖNING, *Theoretische Informatik kurz gefasst*. Wissenschaftsverlag Mannheim, 1992.

[7] M. SIPSER, *Introduction to the Theory of Computation*. 2, Thomson Course Technology Boston, 2006.

[8] T. A. SUDKAMP, *Languages and machines: an introduction to the theory of computer science*. Addison-Wesley Longman Publishing Co., Inc., 1988.

THEORIE-TAG 2017
H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 10–14.

Learning from Leibniz to Understand Thue, Chomsky, and Turing (a Suomi-Romanian Saga)

Erkki Mäkinen Liliana Cojocaru

University of Tampere, Faculty of Natural Sciences
{Erkki.Makinen, Liliana.Cojocaru}@uta.fi

1. A Little Bit of Philosophy

If you want to understand a concept then first understand its philosophy, which is deeply anchored in our intuition of perceiving and judging the things that surround us. A concept or principle has no meaning without its philosophy.

Formal Languages (and Automata) theory is a relatively new computing paradigm, belonging to the 20^{th} century, while its philosophical roots come from Leibniz's works *On the Art of Combinations (and Discovery)* dated from the 17^{th} century. He may be considered the first mathematician to put across the necessity of an *universal language* built from some *primitive axioms* (symbolic assignments) which, through an iterative derivative process (calculating or deduction rules) on combinations of symbols, would lead to more complex structures, thus allowing the *encoding* of human knowledge and providing *logical proofs*.

In his vision everything done by our mind is a *computation*, nothing is *haphazardly*, and therefore everything should be *computable*. Despite the existence of non-decidable problems or Gödel incompleteness theory (a theory of our century), we would rather say that he was perfectly right. The incompleteness property of a system only means that we can build a hierarchy of incomplete computable systems. There is no evidence that this hierarchy would not collapse inside the *constructible Universe*. Isn't it, the Universe, the environment within all problems should be solvable? Otherwise, the *Universe* in which we live would be an incomplete system, which is hardly to believe that may be true. In a few words, *unsolvability* or *incompleteness*, like *time* and *space*, are our own inventions.

Leibniz's philosophical thoughts on an *universal language* and on a *calculating machine* (Step Reckoner) have been materialized during the 20^{th} century, through Thue (or Post) rewriting systems, Chomsky grammar formalism, and Turing machines, giving birth to the so called *Formal Languages and Automata Theory*.

"(Formal) Language is recognized now as an universal paradigm, like *time*, *space*, and *logic*." (Solomon Marcus)

2. And a Bit of History

On its prehistory (1910-1920) Formal Language theory was meant to cope rather with syntactical and structural properties of algebraical structures, such as groups and semigroups, than with linguistic structures (since during those times the algebraic and logic theories were in vogue). The ancestors of nowadays Chomsky grammars were the so called string rewriting systems, developed in turn by Axel Thue and Emil Post in order to handle computations in (combinatorial) group theory, number theory, and mathematical logic.

Noam Chomsky had the brilliant idea to redefine Thue's rewriting rules in order to handle linguistic structures, hence redirecting their applicability to natural language processing (1956). Then there was another crucial mutation that came from Aristid Lindenmayer who refined the semi-Thue systems such that to handle biological structures in living organisms, introducing the so called L-systems, marking thus the beginning of the biological computing era (1968).

As all these scientific challenges would have had almost no significance without their electronic manipulation, an historical milestone was reached then when the grammar formalism, introduced by Chomsky, turned out to be fruitful in defining the syntax of programming languages, hence in compiler constructions. Besides electronics (i.e., hardware), programing languages (i.e., software) proved to be the main tool used to implement the environment under which we live (with all its physical, biological, linguistical, and many other natural properties).

Seen usually as twins, Automata theory and the theory of Formal Languages had officially aroused in 1960. Their ideological roots come from 1936 when Alan Turing invented the first universal formal model of computation - the Turing machine - and from 1946 when Thue-Post rewriting systems were introduced.

Nowadays, Formal Languages and Automata theory stands at the crossroad between Theoretical Computer Science, Computational Linguistics, Computational Biology, Compilers and Programming Languages, and over the years it had sprang its branches throughout the world. It is difficult to estimate which country brought the strongest contribution on this theory, but in almost all countries there were *giants* that did the pioneering steps and impelled further the imagination of generations of researchers. Among them, Finland and Romania showed up with significant results - *jewels that will stand forever*. This was due to legendary and Emeritus Professors: *Arto Salomaa* and *Solomon Marcus*. Their scientific collaboration and friendship that last several decades, and their immeasurable scientific work has been and will forever be relayed to generations of mathematicians and computer scientists. Recently, a biography of Professor and Academician Arto Salomaa has been written, in Finnish, by Emeritus Professor Jukka Paakki, Department of Computer Science, University of Helsinki. The book *Arto Salomaa: Äijän Näköinen Matemaatikko* (*Arto Salomaa: A Grand-Dude Mathematician*) describes the whole academic and research life of Professor Arto Salomaa, along with his fruitful collaboration with Professor Solomon Marcus (and others).

Today, Formal Languages and Automata theory receives priority in the Curricula of the best Finland and Romanian Universities, from undergraduate to master's and doctoral studies.

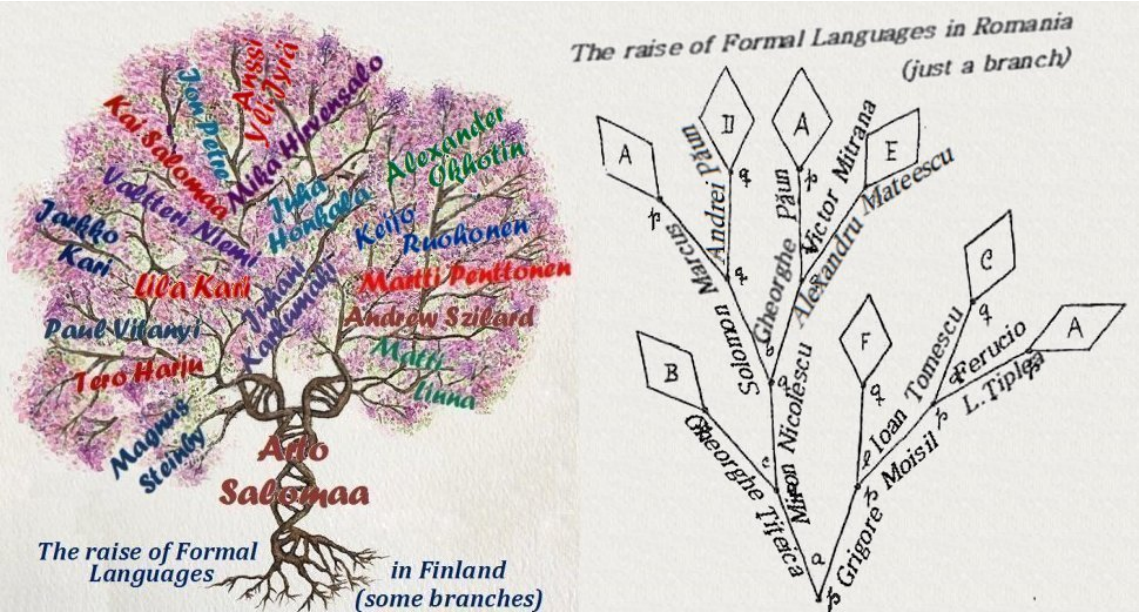
3. Germs in Finland

In Finland, the beginning of formal languages was marked by the inestimable work of Professor and Academician *Arto Salomaa* (Turku Centre for Computer Science and Mathematics, University of Turku, member of the *Academy of Finland* and *Doctor Honoris Causa* of the University of Western Ontario, Canada). His scientific work is synthesized into more than 500 articles in scientific journals and 46 books from which we recall: *Theory of Automata* (1969), *Formal Languages* (1973), *The Mathematical Theory of L-Systems* (1980), *Jewels of Formal Language Theory* (1981), *Public-Key Cryptography* (1990), three-volumes *Handbook of Formal Languages* (1997), and *DNA Computing* (1998).

Professor and Academician Arto Salomaa was the mentor of 25 doctoral students having more than 250 descendants by the Mathematics Genealogy Project. He was in service of science for more than 60 years, and at his honorary age of 83, he is still an active thinker and writer. In recognition of his scientific activity, creativity and originality, academicians and researchers all over the world had brought their honorary homages for his 60^{th} (*Results and Trends in Theoretical Computer Science*), 65^{th} (*Jewels are Forever*), 70^{th} (*Theory is Forever*), 75^{th} (*Symposium*) and 80^{th} (*Arto Salomaa's*) *birthday*.

Professor Arto Salomaa had sealed the research work at Turku Centre for Computer Science and Mathematics, and anywhere in the world, for one hundred years and more. His research activity is carried forward at Turku Centre for Computer Science by the *Automata Theory Group* leaded by Professor Juhani Karhumäki (Fundamentals of Computing and Discrete Mathematics Centre, FUNDIM). The group's research activity is focused on discrete mathematics, combinatorics on words, formal languages and automata, computability and complexity, cryptography and bio-computing. The research projects, at FUNDIM Centre, bring together Emeritus Professor Magnus Steinby, Professors Tero Harju, Kai Salomaa, Jarkko Kari, Ion Petre, Valtteri Niemi, Mika Hirvensalo, Juha Honkala, Alexander Okhotin, and many other researchers and PhD students.

Lectures on Formal Languages and Automata theory have been leaded, during 1970-1980, at the University of Tampere by Emeritus Professor Reino Kurki-Suonio, and during 1980-1990, at the University of Kuopio (now part of the University of Eastern Finland) by Professor Martti Penttonen. During the 70's Formal Languages were also taught at the Technical University of Tampere and the University of Tampere, by Professor Timo Lepistö (a former student of Professor Arto Salomaa) and since 1990, at the Technical University of Tampere, by Professor Keijo Ruohonen. The raise of Computational Linguistics in Finland was due to Professor Kimmo Koskenniemi, the father of *Finite-State Two-Level Morphology*. This is one of the most persuasive application of finite-state transducers in morpho-phonological analysis of natural languages. His work is now carried out, at the Department of Modern Languages, University of Helsinki, by Professor Anssi Yli-Jyrä, who is teaching and performing research on Automata and Formal Languages theory with applications on Finite-State Models as a Fundamental Methodology in Natural Language Processing. Currently, lectures on Formal Languages and Automata theory, with applications in Image Processing and Celular Automata, are taught at Turku University by Professor Jarkko Kari. Research on Bio and Natural Computing, through Formal Languages and Automata theory, are performed at Åbo Akademi University under the guidance of Professor Ion Petre.



4. Germs in Romania

In Romania, as also in Finland, there is a strong tradition to do Mathematics and Computer Science, and this is mostly due to the interdisciplinary oriented work of Professor and Academician Solomon Marcus. His pioneering contributions in fields of Mathematics, Computer Science, Computational Linguistics, and Semiotics made him the most legendary Romanian mathematician of our times. He brought applications of Mathematics, via Formal Languages and Algebraical Linguistics, in Computational Linguistics. Inspired from Chomsky grammars, he introduced in 1969, the so called *Marcus Contextual Grammars* as a generative model of natural languages. For a survey on Marcus contextual grammars, their generative power, and relationships to Chomsky grammars the reader is referred to *Marcus Contextual Grammars* (by Rodica Ceterghi, *Formal Languages and Applications*, Springer-Verlag, 2004).

Professor and Academician Solomon Marcus was also fascinated by the History and Philosophy of Science, and for a more complete History of Formal Languages (than our Bit of History) the reader is referred to his survey on *Formal Languages: Foundation, Prehistory, Sources, and Applications* (*Formal Languages and Applications*, Springer-Verlag, 2004).

Professor and Academician Solomon Marcus was the mentor of 24 doctoral students having more than 100 descendants by the Mathematics Genealogy Project. His scientific work is synthesized into more than 400 articles in scientific journals, conference proceedings, and books, and over 60 books from which we recall: *Introduction Mathematique a la Linguistique Structurale* (1967), *Algebraic Linguistics; Analytical Models* (1967), *La Semiotique Formelle du Folklore. Approche Linguistico-Mathematique* (1978), *Introduccion en la Linguistica Matematica* (1978), *Contextual Ambiguities in Natural and Artificial Languages* (vol.1, 1981; vol.2, 1983), *Language, Logic, Cognition and Communication; A Semiotic, Computational and Historical Approach* (1996). A complete list of his publications can be found on his web page, at the *Simion Stoilow Institute of Mathematics of the Romanian Academy*. His best articles have been gathered into the book *Words and Languages Everywhere* (2007), while the book *Meetings*

with Solomon Marcus (2010) is an honorific volume dedicated to his 85^{th} birthday.

One of the *in memoriam* article of Professor Solomon Marcus was *Grigore C. Moisil: A Life Becoming a Myth*, written in honor of Professor and Academician Grigore Moisil. He is the father of Computer Science in Romania. He did research in the fields of Mathematical Logic, Algebraic Logic, Łukasiewicz-Moisil Algebra, Finite-State and Automata theory. He used the Łukasiewicz-Moisil algebras in Logic and Automata theory. He created new methods to analyze Finite State Automata, with applications in fields of Automata theory and Algebra. Professor and Academician Grigore Moisil had a major contribution in the creation of the first Romanian computers and in the raising of the first generations of Romanian computer scientists. In 1996, he was awarded by exception posthumously the Computer Pioneer Award by the Institute of Electrical and Electronics Engineers Computer Society (Wikipedia).

In Romania, as also in Finland, the Formal Languages and Automata theory is still an active topic in the Curricula of almost all Computer Science and (Applied) Mathematics Faculties, of the best Romanian Universities, such as University of Bucharest, Alexandru Ioan Cuza Universiy of Iaşi, West University of Timişoara, Babeş-Bolyai University of Cluj, Ovidius University of Constanţa. In all these Universities Formal Languages and Automata theory are taught in connection with other adjacent fields, such as Computational Linguistics, Bio and Natural Computing, Petri nets, Computational Complexity theory, Mathematical Logic, and Algebra, for undergraduate, master's, and doctoral students.

Final Notes

This report is just a little part of the Suomi-Romanian Saga on Automata and Formal Languages that has developed over the years. The story is much more complex from both sides.

The former tree in the *Suomi-Romanian Saga* picture is inspired from Adriana Galindo's paintings (on Pinterest), while the second tree is inspired from Axel Thue's paper *Die Lösung eines Spezialfalles eines generellen logischen Problems*, see also *Trees and Term Rewriting in 1910: On a Paper by Axel Thue* by Magnus Steinby and Wolfgang Thomas.

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 15–17.

Automata and Formal Languages in Nigeria

R. O. Oladele

Department of Computer Science, University of Ilorin, P. M. B. 1515, Ilorin, Nigeria.
roladele@unilorin.edu.ng

Zusammenfassung

This paper presents situational report on the teaching/learning of automata and formal languages in nigerian universities. The report is based on personal experience, observation and consultation with some colleagues in other universities.

1. Introduction/Status Report

Nigeria is located on the west of africa and has a population of about 192 million people.There are about 152 universities in nigeria of which 40 are funded by the federal government, 44 are funded by the state government and 68 are owned by private organizations and individuals.

There is a rich theory track in the undergraduate curricula of most of these universities. In particular courses such as; Automata Theory, computability and Formal Languages, Analysis of Algorithms, and Discrete Mathematics and other theory-related courses are in the curricula of almost all these universities. However, it is not clear if all these courses are taught every session by most of these universities.

Automata and Formal Language (AFL) as a course is not taught in some universites, instead compiler construction is taught. In the universities where AFL is taught, it is regarded as a core course which must be offered by all computer science students. Overall, AFL is being taught in most nigerian universities.

At the graduate level (M.Sc. and PhD), less than half of the universities that have graduate programmes teach any of these theoretical courses. In particular, Automata and Formal Languages (AFL) is being taught by a very few of these universities at the masters level.It is pertinent to mention that whenever AFL is taught in these institutions, it is taught as an elective course and there exists very few graduate students offering it.

To the best of my knowledge, among M.Sc. and PhD graduates that are produced in the last 17 years in all nigerian universities, those who wrote their theses/dissertations in the field of Automata and Formal Languages are strictly less than 17. The reason is not unconnected with the fact that there are very few experts doing research in AFL/AFL-related field, specifically there is one Professor in Ahmadu Bello University(Mathematics Department) who has been doing research in AFL-related field since 1999 plus my humble self (a Senior Lecturer in University of Ilorin) who just developed interest in AFL less than two years ago. Another reason is that there are very few or no graduate student opting for research in AFL in these two universities

The table below gives a summary of how AFL fares in the top 10 nigerian universities. The table, among other things, shows if students are taught AFL at the Bachelor (B.SC.), and Masters/Doctorate (M.Sc./PhD) levels. It also shows the number of Lecturers who do active research in AFL, i.e. experts.

Table 1: AFL in top 10 nigerian universities

University	Type	B.Sc.Level	M.Sc./PhD Level	Expert
University of Ibadan	Federal	Yes	Yes	0
University of Nigeria	Federal	Yes	No	0
University of Lagos	Federal	Yes	Yes	0
Obafemi Awolowo University	Federal	Yes	No	0
Covenant University	Private	Yes	No	0
Ahmadu Bello University	Federal	Yes	Yes	1
FUT Minna	Federal	Yes	No	0
University of Ilorin	Federal	Yes	Yes	1
University of Benin	Federal	Yes	No	0
University of Abuja	Federal	No	No	0

2. Challenges of teaching AFL

The challenges of associated with the teaching/learning of AFL in nigerian universites are enumerated below

1. There are no lecturers to teach AFL in many universities: There are only 2 lecturers doing research in AFL/AFL-related area in the whole of nigeria. Worse still, most lecturers are not willing to teach AFL. Consequently, a few ones that are willing to teach are equally employed as visiting lecturers in some other universities.

2. The so-called willing few that accept to teach AFL more often than not gloss over the course hardly covering about 40% of the core topics: The reason for this is not far-fetched, most of these willing few are not passionate/enthusiastic about the course. In fact most of them lack rich understanding/knowledge of what they they teach and they are not willing to learn.

3. Many Students are not willing to learn AFL, they hate it as a course because of it's abstract nature. Most of these students have the misbelief that AFL does not have practical applications.

4. Sometimes majority of the students are not mathematically prepared enough to learn AFL. As such when you teach, you have to slow down, spend a lot of time covering some basics before actually teaching what is in the syllabus. The implication of this is that the actual AFL course syllabus will not be fully covered.

3. Final Remark

Apparently, the state of AFL in Nigeria is worrisome. It is however interesting to state that some students have had their interests in AFL aroused in recent time and this has given birth to some masters theses and some other ongoing projects/research. With this development, it is believed that there is a future for AFL in Nigeria and University of Ilorin is expected to be a major player in realizing this future.

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 18–19.

Ein Plädoyer gegen das Turingband

Klaus Reinhardt[(A)]

[(A)]Institut für Informatik, Universität Halle, Von-Seckendorff-Platz 1, 06120 Halle (Saale)
klaus.reinhardt@informatik.uni-halle.de

Zusammenfassung

In diesem Beitrag wird zur Diskussion gestellt, die Turingmaschine in der Lehre vollständig durch k-PDA's ($k = 2$ für eine Einband-TM) zu ersetzten. Dies wird durch die Einsparung von Zeit und Beschreibungskomplexität begründet.

1. Einleitung

In einschlägigen Lehrbüchern und Skripten wird als Automatenmodell für Typ 0 und Typ 1 Sprachen der Chomsky-Hierarchie die k-Band Turingmaschine (oft $k = 1$ verwendet). Bekanntlich [1] entspricht die Berechnungsmächtigkeit der eines k-PDA mit $k \geq 2$.

Der 2-PDA kann wie der PDA als 7-Tupel $(Z, \Sigma, \Gamma, \delta, q_0, \#, E)$ beschrieben werden, wobei sich nur die Übergangsfunktion ändert zu $\delta : Z \times \Gamma \times \Gamma \mapsto \mathcal{P}_e(Z \times \Gamma^* \times \Gamma^*)$ wobei $\Sigma \cup \{\#\} \subset \Gamma$. Mit der Startkonfiguration $(q_0, x\#, \#)$ auf Eingabe x und der Konfigurationsübergangrelation $\vdash$ mit $(q, a\alpha_1, b\alpha_2) \vdash (p, \beta_1\alpha_1, \beta_2\alpha_2)$ für $(p, \beta_1, \beta_2) \in \delta(q, a, b)$ verallgemeinert eine Rechnung direkt die Rechnung eines PDA. Dabei wird nur das Eingabeband ersetzt durch den ersten Keller der das Eingabewort x (nur das Kelleranfangszeichen # kommt hinzu) enthält und auf den nichts geschrieben wird.

Die Verallgemeinerung zum k-PDA ist kanonisch.

Auch Akzeptieren mit leeren Kellern anstelle von Endzuständen ist möglich.

2. Vorteile

1. Anstelle der Einführung des neuen Kozeptes "Band" wird das bereits erlernte Konzept "Keller" vertieft.

2. Beispielautomaten haben weniger Zustände als die entsprechende Turingmaschine und sind leicht nachzuvolziehen.

3. Der LBA ist bereits durch die Bedingung $|\beta_1\beta_2| \leq 2$ für alle $(p, \beta_1, \beta_2) \in \delta(q, a, b)$ vollständig definiert.

4. Die Simulation von Typ-0 und Typ-1 Grammatiken vereinfacht sich deutlich (kein Verschieben von Halbbandinhalten mehr).

5. Die Simulation eines k-DPDA durch einen 2-DPDA ist leichter zu beschreiben als die Simulation einer k-DTM durch eine 1-DTM, da das Aufsammeln der obersten Kellerzeichen in geordneter Reihenfolge erfolgt.

6. Leichte Einsparung auch bei der Gödelisierung, da die Kodierung der Bewegungsrichtungen entfällt.

7. Leichte Vereinfachung der Simulation eines k-DPDA durch ein GOTO-Programm gegenüber der einer DTM.

8. Der offline-k-PDA, der für die logarithmisch platzbeschränkte Reduktion benötigt wird, ist ebenfalls leicht durch die Bedingungen $\beta_1^R\beta_2 = a_1a_2$ und $|\beta_k| \leq 1$ für alle Übergänge $(p, \beta_1, \beta_2, ...\beta_k) \in \delta(q, a_1, a_2, ...a_k)$ definiert. (Damit kann für die Funktionsberechnung nur der Platzverbrauch in Keller 3 bis $k-1$ betrachtet werden.)

3. Nachteil

Aufwand bei der Umstellung von Lehrmaterial.

Literatur

[1] J. E. HOPCROFT, R. MOTWANI, J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 20–28.

Teaching Theoretical Computer Science with Python

Heinz Schmitz[(A)]

[(A)]Hochschule Trier, Fachbereich Informatik, Schneidershof, D-54293 Trier, Germany
H.Schmitz@hochschule-trier.de

Abstract

We show how teaching classical content from Theoretical Computer Science (TCS) can be combined with an introduction to the PYTHON programming language within the same course as part of a Bachelor program in Computer Science. Our main motivation to do so is to obtain executable versions of TCS models and to introduce PYTHON in a thorough way for follow-up courses. Due to PYTHON's reduced syntax and hence pseudocode-like appearance, TCS models can be represented and also executed in a concise way very close to the formal definitions. We outline the resulting course structure and show how PYTHON can be interlaced with a typical sequence of lectures on TCS while preserving the mathematical nature of the course. We believe that this combination can help students to bridge the putative gap between theoretical and more practical aspects of CS, and that it provides a more seamless integration of TCS into the overall curriculum.

1. Introduction

Introductory courses on Theoretical Computer Science are a fundamental part of most study programs in Computer Science (CS) at German universities, and there is a broad agreement what subjects should usually be covered [6]. These subjects can be roughly termed as automata theory and formal languages, computability and complexity theory, and each time mathematical models serve as abstract representations of some computation resources and processes. Students are expected to develop the capability to design and analyse these models in order to understand the fundamental relations that delimit and determine the field. But also more practical learning outcomes are wanted: Formalization abilities and rigorous thinking are seen as important ingredients to high-quality work results throughout entire CS.

As has been frequently reported in the literature, students often have difficulties following in particular the TCS courses, resulting in substantially high failure rates and unsatisfactory learning outcomes. So various approaches have been suggested to improve this situation, among them strengthening the historical context [1], visualizations of mathematical models [5] or a cognitive apprenticeship approach [4]. The latter is a concept that addresses the whole process of teaching and learning, since there does not seem to be a single and stable set of influencing factors. We contribute to this ongoing discussion with an add-on to the *content* of such a course but within the bounds of mandatory TCS subjects.

Our impression is that students see TCS models mainly as some 'artificial' objects with no obvious links to practical areas of CS, although references to these other areas and to forthcoming courses are usually given in a TCS lecture. On the other hand, (small) models of programming languages are treated as models for computation which have a clear and strong connection to what students experience elsewhere in CS. Based on this simple observation, we derive the following ideas to help students bridge the putative gap between theoretical and more practical aspects of CS:

- We use a small subset of a widespread programming language as a WHILE language when teaching computability. This has been common practice in TCS courses before, e.g., RIES as a PASCAL-like language in [7]. Here we choose PYTHON[1] to make use of it's reduced syntax and hence pseudocode-like appearence. Together with an almost math-like notation for expressions and sets this makes PYTHON a good candidate for TCS purposes. Moreover, it can be expected that PYTHON plays it's role elsewhere in the CS curriculum.

- After that, we use this WHILE language (with some extensions) consequently to represent and execute all forthcoming TCS models in the course. This also includes implementations of algorithms derived from all constructive proofs. So on one hand the (extended) WHILE language is immediately applied to some 'real' computation tasks, and on the other hand students can investigate all TCS models and proofs by executing them in a present-day language.

- Parallel to the treatment of different TCS models, the WHILE language is stepwise extended by basic data types and their operations in order to allow more convenient programming. Also runtime analysis, first introduced for the plain WHILE language, is carried along these extentions by expanding the cost model and definitions of input lengths accordingly. So the TCS models and algorithms in PYTHON can be immediately judged by their efficiency by both, theoretical bounds and practical experiences. As a consequence, the introductory TCS course in not only a basis for CS in general and for advanced TCS courses in particular, but it can also serve as a fundament for follow-up courses on algorithms when based on PYTHON.

We believe that these elements result in a more seamless integration of TCS into the overall CS curriculum, and hopefully less students look at it as something that they just have to get along with.

The course we outline in the next section was develped over the last decade and has some accompanying features. It is given in a series of 15 lectures each 90 minutes per week, accompanied by weekly exercise sessions of the same duration with up to 30 students where homeworks need to be presented. All lectures are recorded as *screencasts* and students are asked to work through them before the weekly *tutorial* which is a session given by the instructor instead of the lecture. Here questions can be asked, additional examples and live-programming are presented and solution hints to more complicated exersices are discussed. An online *quiz* with about 500 true/false-questions about the lecture contents is intended to activate students. The main lecture

[1] www.python.org

material is a script where proofs and examples are added by handwritten tablet input during each lecture. Students are invited to turn it into their *personal workbook* by adding their own comments.

2. Course Outline

The content can be roughly sectioned into basics (lectures 1 to 3), computability (4,5), runtime analysis (6,7), finite automata and regular languages (8 to 11), contextfree languages (12,13) and complexity (14,15). There is no particular lecture on PYTHON since language elements are introduced en passant when needed. The teaching order is a result of the ideas mentioned before. The early treatment of computability together with the introduction of a PYTHON subset as a WHILE language is needed to provide a model of computation that can be immediately applied in subsequent lectures. For the same reason runtime analysis of these WHILE programs and their extensions are considered next, so this machinery can also be applied to the algorithms presented later in the course. On the other hand, complexity classes are only introduced towards the end of the course when various programming experiences have been made in previous exercises. Next we give some keywords for the content of each lecture and show how PYTHON is interlaced. We also mention some exercises that can be assigned in addition to usual TCS paperwork homeworks.

1. *Introduction.* Syllabus, organizational notes; formulation of theorems and basic proof forms.

   The PYTHON part here is only to install all needed software and get an IDE running (we use PYTHON 3.x, Eclipse[2] and the PyDev[3] plugin).

2. *Induction.* Induction on $\mathbb{N}$, inductive definitions, structural induction, examples of recursive functions.

   Additionally, we give an informal introduction to the PYTHON shell and we declare and execute a simple recursive function. It's correctness proof serves as an example for the application of the induction principle in a CS context. Moreover, the set of feasible arithmetic expressions as used later in the WHILE language are among the examples for inductive definitions, and some easy properties of these expressions (e.g. numbers of opening and closing brackets coincide) are shown by structural induction.

3. *Words and Formal Languages.* Alphabets, words, formal langauges, word problem, basic language operations.

   In this lecture, we also introduce PYTHON string and set types together with basic operations for both of them. So finite sets of words can be explicitely handled, and differences between TCS and PYTHON notations are shown, e.g.

[2] http://www.eclipse.org
[3] http://www.pydev.org

	TCS notation	PYTHON
words, empty word	$w = 01101$, ε	`w = '01101', ''`
alphabets	$\Sigma_1 = \{0,1\}, \Sigma_2 = \{a,b,c\}$	global
length, concatenation	$\lvert w\rvert$, wv, w^5	`len(w)`, `w + v`, `w*5`
indexing, subwords	$w = a_0a_1a_2$, $a_i \cdots a_j$	`w[0]w[1]w[2]`, `w[i:j]`
sets, empty set	$A = \{\varepsilon, 0, 00\}$, $E = \emptyset$	`A = {'','0','00'}`, `E = set()`
membership	$x \in A, x \notin A$	`x in A`, `x not in A`
language operations	$A \cup B, A \cap B$	`A | B`, `A & B`
	$\overline{L}$, L_1L_2, L^k, L^*, L^R	no such thing

Additional exercises cover implementations of language operations in PYTHON where finiteness is preserved.

4. WHILE *Programs.* Models of computation, syntax and semantics of a WHILE language, examples.

   We introduce a carefully chosen subset of PYTHON as a model for computation so students immediately have the PYTHON shell as an execution environment available. The syntactic elements are inductively introduced together with their semantics and comprise:

   Constants $[-](0|(1|\ldots|9)(0|\ldots|9)^*)$

   Identifiers $(a|\ldots|Z)(a|\ldots|Z|0|\ldots|9)^*$

   Expressions constants, variables
   `(a+b)`, `(a-b)`
   `f(b1,...,bm)`

   Conditions `(a==b)`, `(a!=b)`, `(a>b)`, `(a<b)`, `(a>=b)`, `(a<=b)`, `(not c)`, `(c1 or c2)`, `(c1 and c2)`

   Assignment
   ```
   a = b
   ```

   Cond.Stmts
   ```
   if c:
       s1
   else:
       s2
   ```
   ```
   if c:
       s
   ```

   Composition
   ```
   s1
   s2
   ```

   Loops
   ```
   while c:
       s
   ```
   ```
   for i in range(a1,a2):
       s
   ```

   Functions
   ```
   def f(a1,...am):
       s
       return a
   ```
   ```
   def f(a1,...am):
       return a
   ```

   Programs
   ```
   f1
   ...
   fm
   ```

   Typical exercises ask for partial functions φ_P on $\mathbb{ZZ}$ computed by a given WHILE program P, and vice versa. For the latter, students need to provide implementations that pass at least the given test cases.

5. *Computability.* Class of WHILE-computable functions, equivalence to other models of computation (w/o proof), Turing-completeness, Church-Turing-hypothesis; existence of incomputable functions by counting (diagonalization); decidability, examples of undecidable sets.

   Additional practical exercises for this lecture treat algorithms for characteristic functions of some example problems, a WHILE implementation of the Ackermann function and (pseudo-) WHILE algorithms for closure properties of recursive sets.

6. *Runtime Analysis.* Types of cost models, uniform model for WHILE, step function, runtime function, input length, asymptotic classes and their properties, runtime analysis of WHILE programs.

   The main focus in the additional exercises is to analyse the runtime of given WHILE programs. However, students also gain hands-on experience in the significance of theoretical bounds, e.g., when experimentally comparing the naïve WHILE program for multiplication by successive summation with a more clever algorithm for it.

7. *Python Programs.* Specification of computational problems, classical examples like KNAPSACK and pattern matching in strings; extension of WHILE programs by data types `str`, `set`, `list` and `dict`, and their operations; expansion of the cost model, definition of input length for the new data types, examples of runtime analysis.

   More classical decision and optimisation problems are introduced in the exercises, e.g., BIN PACKING and SOS. Exhaustive search algorithms and their runtime analysis using the new data types need to be derived.

8. *Deterministic Finite Automata.* Examples and definition of DFAs, extended transition function and accepted language; decision problems for DFAs; extended example of search automata for string matching.

   Together with the formal definitions, DFAs are immediately represented in PYTHON. All ingredients needed to do so have been introduced before. We contrast an example DFA with its PYTHON representation.

state set $Q = \{q_0, q_1, q_2\}$
alphabet $\Sigma = \{0, 1\}$
accepting states $F = \{q_2\}$
transition function δ as:

q	a	$\delta(q,a)$
q_0	0	q_1
q_0	1	q_0
q_1	0	q_1
q_1	1	q_2
q_2	0	q_2
q_2	1	q_2

DFA $A = (Q, \Sigma, \delta, q_0, F)$

```
Q = {0, 1, 2}
Sigma = {'0', '1'}
F = {2}
delta = {}

delta[0,'0'] = 1
delta[0,'1'] = 0
delta[1,'0'] = 1
delta[1,'1'] = 2
delta[2,'0'] = 2
delta[2,'1'] = 2

A = [Q, Sigma, delta, 0, F]
```

Similarly, the inductive definition of the extended transition function has an immediate recursive implementation in PYTHON.

ext. trans. function $\widehat{\delta} : Q \times \Sigma^* \to Q$ as

- $\widehat{\delta}(q, \varepsilon) = q$, and
- $\widehat{\delta}(q, wa) = \delta(p, a)$ if $p = \widehat{\delta}(q, w)$

```
def delta_hat(delta, q, v):
    if v  == '': return q
    w, a = v[:-1], v[-1]
    p =  delta_hat(delta, q, w)
    return delta[p, a]
```

Finally, the definition of the accepted language has an executable counterpart in PYTHON.

$$L(A) = \{w \in \Sigma^* \mid \widehat{\delta}(q_0, w) \in F\}$$

```
def run_dfa(A, w):
    [Q, Sigma, delta, q0, F] = A
    return delta_hat(delta, q0, w) in F
```

Observe that no more code is needed to run a DFA on some input word. Students benefit from this in the exercises when the behaviour of self-designed automata for given languages can be inspected. Moreover, algorithms for emptiness and finiteness tests for a given DFA can be programmed in PYTHON. An example of a more advanced task is to implement the construction of the search automata for string matching as treated in the lecture.

9. *Nondeterministic Finite Automata.* Examples and definition of NFAs, extended transition function and accepted language; equivalence of DFAs and NFAs, closure properties.

   Again, NFAs are represented in PYTHON and also all other definitions are carried over. Similarly few code is needed to obtain executable NFAs. The runtime analysis of the respective `run_nfa` function reveals the price to pay for less states and easier design. Evident exercises cover implementations for the constructive proofs given by the powerset construction and for the closure properties.

10. *Regular Expressions.* Syntax and semantics of regular expressions, conversion to NFAs, construction of a regular expression for given DFA with a Floyd/Warshall-like algorithm.

    Although regular expressions are treated only as flat strings in PYTHON, they will be parsed in later exercises. Here regular expressions are used to specify valid WHILE constants and identifiers, and automata need to be obtained to recognize them by the methods introduced in the lecture. Additionally students are asked to implement the construction of the regular expression from a given DFA by dynamic programming. Here they gain first-hand experience about what an exponential growth of the output size looks like.

11. *Regular Languages.* Proof and applications of Pumping Lemma. Summary of lectures 8 to 11.

    Students have implemented and experienced up to now a collection of algorithms transforming different representations of regular languages. This own experience adds to the motivation of asking for which languages these nice things are not possible. The summary additionally emphasizes the derived runtimes of simulation and transformation algorithms, and reveals trade-offs between regular language representations.

12. *Contextfree Languages.* Definition and examples of contextfree grammars, derivations, from DFAs to CFGs; PYTHON data type `tuple`, equivalence of left-, right-derivations and parse trees, ambiguity; ε-free grammars and their construction.

    The representation of CFGs in PYTHON is again chosen close to the formal definitions, where each production is just a tuple of the left and right side. This may also apply for other types of grammars.

terminals $\Sigma = \{0, 1\}$	`Sigma = {'0', '1'}`
non-terminals $N = \{S\}$	`N = {'S'}`
set of productions P:	
$S \rightarrow \varepsilon$	`P = {  ('S', ''),`
$S \rightarrow 0S1$	`  ('S', '0S1') }`
CFG $G = (\Sigma, N, P, S)$	`G = [Sigma, N, P, 'S']`

    Based on this, a leftmost derivation step for a given production is easy to implement.

if $p = (A \rightarrow \beta)$

and $\alpha = wA\gamma$

$\alpha \Rightarrow_l w\beta\gamma$

```
def derive_left(alpha, p):
    (A, beta) = p
    if A in alpha:
        i = alpha.index(A)
        w, gamma = alpha[:i], alpha[i+1:]
        return w + beta + gamma
    else:
        return alpha
```

    Additional exercises cover the implementation of turning a PYTHON representation of an arbitrary DFA into the respective CFG, or to return the set of all words that can be produced with at most k leftmost derivation steps for a given CFG.

13. *An All-round Parser.* Chomsky normal form, CYK algorithm, correctness and runtime; parsing arithmetic expressions of WHILE programs.

    The lecture demonstrates the small step from the inductive definition of subproblems for the CYK method to an executable implementation by dynamic programming. Additional exercises ask for parsing WHILE expressions including function calls, where results from previous exercises on valid WHILE identifiers need to be reused, and for parsing regular expressions over a given alphabet.

14. *The Classes P and NP.* Definition of P, FP and EXP, closure properties of P, common problem structure of SOS, TSP and others; definition of NP, inclusions from P and to EXP.

    Here NP is defined via length-bounded certificates and poltime verification, so in additional practical exercises students can implement these verifications for given solution candidates and convice themselves of their poltime by runtime analysis. Moreover, suitable iterators are available in PYTHON's `itertools` in order to obtain canonical exhaustive search algorithms in a straightforward way, while runtime analysis of these

PYTHON implementations shows membership in EXP. Here is the entire algorithm for the SOS problem.

for $a = (a_0, \ldots, a_{m-1}), k$

exists $I \subseteq \{0, \ldots, m-1\}$ with $k = \sum_{i \in I} a_i$?

```
from itertools import product
def sos_exhaustive(a,k):
    m = len(a)
    for I in product((0,1),repeat=m):
        if k==sum(a[i] for i in range(m)
                              if I[i]):
            return 1
    return 0
```

15. *NP-Completeness.* Poltime-many-one-reducibility, example reductions, closure of P and NP under $\leq_m^p$, notion of NP-completeness, examples (w/o proof), discussion of P/NP-question.

    Typical exercises ask for reductions between similar problems extended by implementations of these reduction functions.

3. Outlook

Time is the resctrictive factor when including additional topics, and other things have to be left out. In our case, we moved the dicussion of Turing machines and proofs of their equivalence to other models to later courses. The same holds for a self-contained NP-completeness proof and the other levels of the Chomsky hierarchy. We found that adding PYTHON implementations also in these later courses is helpful there too, and not much additional effort is needed after this introductory course, e.g., to obtain a straightforward representation and execution of Turing machines or the implementation of their Gödelization. Another follow-up course that is now well-prepared may deal with syntactic analysis, where discussion and representation of PDAs can be easily motivated and combined with the development and implementation of parsing algorithms – based on PYTHON and the introduced cost model. The same is true for a course on basic algorithms that can follow seamlessly [3, 2]. The introductory TCS course described here does not only allow for additional exercises, but it also yields subjects for practical but theory-based student projects or theses that are well-prepared by this course. E.g., this could be the development of a WHILE parser that adds to each program a counter for computation steps according to the cost model, or a PYTHON module to visualize and interact with the transition diagrams of the automata models, or a collection of implementations of reduction functions that make use of some solver package for satisfiability or linear optimization in the background, only to name a few.

Acknowledgements

This teaching concept has been jointly developed and intensely discussed with Christian Glaßer, Universität Würzburg, Germany. We are also grateful to Christian Reitwießner for pointing out the advantages of PYTHON to us.

References

[1] C. I. CHESÑEVAR, M. P. GONZÁLEZ, A. G. MAGUITMAN, Didactic strategies for promoting significant learning in formal languages and automata theory. *ITiCSE* (2004), 7.

[2] T. HÄBERLEIN, *Praktische Algorithmik mit Python*. Oldenbourg Verlag, 2012.

[3] M. HETLAND, *Python Algorithms*. Apress, 2010.

[4] M. KNOBELSDORF, C. KREITZ, S. BÖHNE, Teaching theoretical computer science using a cognitive apprenticeship approach. *SIGCSE* (2014), 67–72.

[5] S. H. RODGER, B. BRESSLER, T. FINLEY, S. READING, Turning automata theory into a hands-on course. In: *the 37th SIGCSE technical symposium*. ACM Press, New York, New York, USA, 2006, 379.

[6] UNKNOWN AUTHOR, *Empfehlungen für Bachelor- und Masterprogramme im Studienfach Informatik an Hochschulen (Juli 2016)*. Gesellschaft für Informatik e.V., 2016.

[7] K. W. WAGNER, *Theoretische Informatik – Eine kompakte Einführung*. second edition, Springer, 2003.

THEORIE-TAG 2017

H. Fernau (Herausgeber): Theorietag 2017, Bonn, 18. – 22.9.2017
Universität Trier, Technischer Bericht, S. 29–32.

Teaching of Automata and Formal Languages - Indian Scenario

D. G. Thomas

Department of Mathematics, Madras Christian College, Chennai - 600059, India
dgthomasmcc@yahoo.com

The awareness of higher education in India is very high as almost every student is completing school education and pursuing higher education. Most of the students take up under graduate courses either in science or engineering. Any under graduate program in computer science, computer applications and mathematics can offer a course in formal languages and automata theory (FLAT).

1. **Higher Education in India**

   There are around 800 universities/institutions managing higher education in India. Many of them are funded by government and others by private managements. The following table shows the statistics of universities and institutions available in India as per the All India Survey on Higher Education for the year 2015-2016.

   No. of Universities: 799 (Private Universities: 277)

   No. of Colleges: 39071

Type of University	Number of Universities
Central Open University	1
Central University	43
Deemed University Government	32
Institution Under State Legislature Act	5
Institution of National Importance	75
Deemed University - Private	79
State Private University	197
State Open University	13
State Public University	329
State Private Open University	1
Deemed University - Government Aided	11
Others	13
GRAND TOTAL	799

2. **Institutions of National Importance**

   There are around 70 technical institutions and a few institutions for science and technology of national importance funded by Government of India. To mention a few:

- Indian Institute of Technology (IIT): 16
- National Institute of Technology (NIT): 31
- Indian Institute of Information Technology Design & Manufacturing (IIITDM): 23
- Tata Institute of Fundamental Research (TIFR), Mumbai
- Indian Statistical Institute (ISI), Kolkata
- Indian Institute of Science (IISc), Bengaluru
- Institute of Mathematical Sciences (IMSc), Chennai

3. **Theoretical Computer Science (TCS) & Formal Languages and Automata Theory (FLAT) Notable Institutions:**

   Among the institutions of science and technology, the following are notable for offering courses in TCS and FLAT in under graduate (B.Sc., B.C.A, B.E and B.Tech) and in post graduate levels (M.Sc., M.C.A., M.E., M.Tech and M.Phil.), and for pursuing research degree (Ph.D.).

   - Tata Institute of Fundamental Research (TIFR), Mumbai
   - Indian Statistical Institute (ISI), Kolkata
   - Indian Institute of Science (IISc), Bengaluru
   - Institute of Mathematical Sciences (IMSc), Chennai
   - Chennai Mathematical Institute (CMI), Chennai
   - Madras Christian College (MCC), Chennai

4. **Madras Christian College (MCC), Chennai**

   The Madras Christian College which is one among the top 10 colleges in India was founded as a school in George Town, Chennai in 1837. It has moved to the present campus of 365 acres in Tambaram, Chennai in 1937. The year 1978 was path-breaking as MCC became one of the earliest colleges to be granted the status of autonomy. This status enabled the college to introduce new courses and innovative curricular enhancements. At this juncture, FLAT was introduced as a course in B.Sc. and M.Sc. by the Department of Mathematics, MCC. The department has grown into a high level research center in FLAT. At present there are 32 Academic departments in MCC with 304 teaching staff catering to the needs of 7200 students coming from semi-urban background comprising 48% women.

5. **The school of Siromoney**

   A decade before autonomy, Dr. Rani Siromoney started working on her Ph.D. in the field of FLAT. She has built a very strong research group in FLAT which continues to be active till date. She has guided 4 Ph.D. students in FLAT. They are: Prof. Kamala Kirthivasan (IIT-M), Prof. K.G. Subramanian (MCC), Prof. V. Rajkumar Dare (MCC) and Prof. K. Rangarajan (MCC). There are many grand doctoral students of Dr. Rani Siromoney. To mention a few: Dr. Meena Mahajan (IMSc), Dr. D.G. Thomas (MCC), Dr. P.J. Abisha (MCC) and Dr. T. Robinson (MCC) and Dr. R. Rama (IIT-M).

6. **Books Written/Edited**

   To enhance and enrich the teaching of FLAT, the books written/edited by Siromoney school are given below:

   (a) Rani Siromoney - Formal Languages and Automata, CLS, 1984

   (b) Kamala Kirthivasan and R Rama, Introduction To Formal Languages, Automata Theory and Computation, Pearson, 2011

   (c) K G Subramanian and K Rangarajan - Formal Methods, Languages and Applications, World Scientific, 2006

   (d) D G Thomas and P J Abisha - Cryptography, Automata and Learning Theory, Narosa, 2011

   (e) D G Thomas and T Robinson - Automata, Graphs and Logic, Narosa (in press), 2017

7. **MCC - Courses Taught**

   - In M.Phil. Programme, a course on Theory of Computation and Graph Theory is offered. The topics taught are: Turing Machines, Decidability, Reducibility and Time Complexity. The text book prescribed is: Introduction to the Theory of Computation, 2^{nd} Ed., authored by Michael Sipser.
   - In M.Sc. Programme, a course on Formal Languages and Automata is offered. The topics taught are: Finite Automata, Regular Grammars, Properties of Regular Languages, Context-Free Languages, Pushdown Automata, and Properties of Context-free Languages. The text book prescribed is: An Introduction to Formal Languages and Automata, 4^{th} Ed., authored by Peter Linz.
   - In M.Sc. Programme, another course on Theory of Computation is offered. The topics taught are: Turing Machines, Other Models of Turing Machines, A Hierarchy of Formal Languages and Automata, Limits of Algorithmic Computation and an Overview of Computational Complexity. The text book prescribed is: An Introduction to Formal Languages and Automata, 4^{th} Ed., authored by Peter Linz
   - In B.Sc. Programme, a course on Formal Languages and Graph Theory is offered. The topics taught are: Phrase - Structure Languages, Closure Properties, Normal forms of CFG, Properties of CFLs and Finite State Automata. The text book prescribed is: Formal Languages and Automata, 2^{nd} Ed., authored by Rani Siromoney.

8. **IIITDM - Course Taught**

   With the objectives of training students to design various phases of compiler such as Lexical Analyzer, Syntax Analyzer, Semantic Analyzer, Intermediate Code Generator, Code Optimizer and Code Generator and understanding the applications of FSA & PDA in Compiler Design, the course on Automata & Compiler Design is offered in B.E.(Computer Engineering) programme. The text book prescribed is: Principles, Techniques and Tools, authored by Alfred Aho, Ravi Sethi and Jeffrey Ullman.

9. **IIT Madras - Course Taught**

   With the objective of providing a formal connection between algorithmic problem solving and theory of languages and automata and developing FLAT into a mathematical view towards algorithmic design and in general, computation itself, the course Languages, Machines and Computation is offered in B.E. (Computer Science Engineering) Programme. The topics taught are: Finite Automata & Regular Languages, Non-determinism & Regular Expressions, Grammars & Context-free Languages (CFLs) and Turing Machines & Computability. The text books prescribed are: Automata and Computability, authored by Dexter C. Kozen, 2007 and Introduction to Automata Theory, Languages and Computation, authored by Hopcroft, Motwani and Ullman, 3^{rd} Ed., 2006

10. **FLAT - Challenges & Needs**

    Challenges:

    - Students to inculcate Mathematical thinking skills
    - Students to develop intuitive ideas for problem solving
    - Students to take up projects

    Needs:

    - Objectives to be prescribed
    - More applications to be shown
    - Software/Teacher's resources to be prepared
    - Lab activities/projects to be developed
    - Student-friendly textbooks to be prescribed
    - Workshops for course teachers of neighbouring institutions to be organized