Back to BASIC in Compiler Construction
(short position paper)

Stefan Gruner

Department of Computer Science, University of Pretoria, South Africa
sg@cs.up.ac.za

Abstract. This short-paper offers an experience report about a successful way of giving an introductory compiler construction course to 3rd-year undergraduate students. Because the in-depth-presentation of compiler construction has nowadays become rather seldom at South African universities, this short-paper is intended to serve as motivation and recipe for the topic's (re)introduction at other institutions of tertiary education.

Keywords: Compiler Construction, 3rd-year Undergraduate, BASIC, Experience Report, Tertiary Education, Computer Science.

1 Motivation and Related Work

In bygone days there was a sharp distinction between *vocational* tertiary education in 'technikons' —which emphasided employability and industry-readiness for their students— on the one hand, and *academic* tertiary education at universities —which emphasised scientificness or scholarliness and which were by-and-large free from industrial interference— on the other hand. Recent trends in tertiary education both nationally and internationally have increasingly blurred the old line of separation between technikons and universities, whereby the (former) technikons are now striving for higher scholarly reputation (thus competing against the classical universities); the universities are now increasingly emphasising and advertising the employability and industry-readiness of their students (thus competing against the former technikons); like the curricula of the former technikons, also the curricula taught at universities are now increasingly influenced by the commercial industry (often via external advisory boards). Under these circumstances and a consequence of this line-blurring trend, which is well-documented in large quantities of literature on higher education —for only three example see [12][13][16]— some universities nowadays might feel tempted to dilute (if not entirely abolish) a number of classical courses that are now being regarded as 'too theoretical', 'not practical enough', or 'not industrially relevant'. Whereas the classical university of bygone days confronted its students with difficult theoretical *science* from day one onwards, nowadays trend is to first *train* students in industrially applicable *skills*, and to let the science follow only later at post-graduate level. Indicative of this trend seems to be also the nowadays mushrooming usage of the phrase *"teaching* and *learning"* (instead of

lecturing and studying) in *tertiary* education management jargon which seems to signify an ongoing *schoolification* of tertiary education at universities.[1]

The topic of this short position-paper —*compiler construction*— cannot be separated from those above-mentioned general trends. In South Africa, for example, only few of the country's tertiary education institutions offer courses on this topic at all, and also internationally the curricular relevance of compiler construction has been disputed [7][11][22]. However, the difference between learning programming and studying compiler construction is similar to the difference between learning how to drive a car and studying automotive engineering for the sake of car-construction: learning programming and learning to drive a car are simple enough that teenagers can do that at secondary school before entering university, whereas science-based tertiary education is necessary to master the challenges of compiler construction (i.e.: *making* new programming languages rather than merely using existing ones) as well as automotive engineering. Beheld from this perspective it should be clear that computer-*scientific* topics like compiler construction ought not to vanish from tertiary computer science curricula — though this has appened in fact from place to place. Indeed: if (as mentioned above) also universities are nowadays becoming increasingly employability-skills-oriented, and if the local IT industry mainly wants JAVA programmers, then why bother 'learning' compilers at university? Such a point of view, however, would ignore the educational benefits of compiler construction as *computer science in a nutshell*, in which the threads of many sub-topics of computer science (e.g.: automata theory, algorithms and data structures, principles of software engineering, operating systems, artificial intelligence methods of optimisation, and the like) are woven together [9]. In the undergraduate curriculum, compiler construction is one of the few theoretically solid topics by means of which the the *scientific-ness* of computer *science* (as opposed to computer engineering or software 'crafting') can be convincingly demonstrated. However: even the rather small minority of those students, who are still coming to university with a classical scientific outlook [10], do not find it easy to imagine a compiler's theoretical concepts and inner workings.

In this short-paper report I briefly describe *qualitatively* what I have done to provide a fruitful study experience for my compiler construction students under those above-mentioned circumstances. Thereby my pedagogical efforts were particularly aimed at making the students intuitively see an actually running program after its compilation — i.e.: to provide the students with some joyful personal 'heureka!' experiences which are arguably important from a pedagogical point of view. On-the-fly I also tried to emphasise the computer-*scientific* foundations of compiler construction wherever possible (for example: by pointing out issues of *undecidability* on various occasions) with the aim of nurturing a scientific world-view also among those (many) students who are nowadays at university just for the above-mentioned employability skills [10]. To those students I tried to advertise the benefits of a scientific world-view rather instrumentalistically with the message: "*solid science will help you to do a better job*".

[1] I still remember my chemistry teacher: "at school you learn; at university you *study!*"

Academic literature on compiler *education* dates back to the mid-1960s. Due to the wide-spread standardisation of compiler construction, publications on this topic appear in only irregular frequency and not in large numbers. In *"a new approach to teaching a first course in compiler construction"* from 1976 it was recommended to replace one large semester project by several smaller independent mini projects [18]; I have followed that route. An *"emulator"* approach to code generation was advocated already in 1977 [15]; I have followed that route as well. The Student Programming Language SPL used in my course is similar to the project language of [3]. The (disputed) importance of compiler construction within the computer science curriculum was emphasised again during the mid-1990s [2]. The topic-specific difficulties with which compiler construction students are typically confronted are summarised in [19]: for similar reasons the students of [4][21] were given a partly pre-fabricated compiler environment to 'play' with; this rather shallow educational 'use and play' approach, however, is not advocated by me. The often-repeated opinion about compiler construction being allegedly 'out-dated', 'too old-fashioned' or 'irrelevant' for nowadays curricula was analysed and discussed in [11] wherein also some new topics for advanced compiler construction courses were presented. A similar modernisation attempt was presented in [22] — albeit with the danger that all those *"exciting"*[22] software-controlled robots might only distract the students from the scientific essentials of the topic. In [7] the topic's relevance question was raised, too, and yet another tool kit in support of new approaches to compiler construction was presented. A nice and insightful personal letter on this topic was written by the main author of the seminal 'dragon book' [1]. Whereas the practical part of my course ends with the generation, optimisation and variable-liveness analysis of *intermediate* code, whereby the generation of target machine code is merely lectured, the authors of [14] have created a small and simple set of pseudo machine code instructions by means of which their students can practice also the final compiler phase of *target* code generation. However, similar to my approach at intermediate code level, the pseudo machine code of [14] is *emulated*, too.

2 Experience Report

The introductory compiler construction course described in this short-paper is a 3rd-year course in one academic half-year (semester). Two lectures took place per week with 50 minutes duration per lecture. Moreover, eight practicals had to be done and demonstrated. The work-time per practical was approximately 7–10 days. No additional recapitulation- or tutoring-lessons (outside the regular lectures) were provided to the students, who thus had to be highly self-sufficient, self-responsible, self-motivated, and diligent. All in all, each student implemented a fully operational compiler from scanner (lexer) and parser via static semantics analyser (name-scope analysis, DECL-APPL analysis, value-flow analysis and type-checking) to intermediate code generation (translation to BASIC), followed by some intermediate code optimisation (on the generated BASIC code) as well as

variable liveness analysis and variable dependency analysis (yielding a coloured dependency graph) also on the generated BASIC code.

Mogensen's book [17] provided the conceptual foundations, whereby the students had rather little prior knowledge from their previous study-years 1–2 in theoretical informatics, i.e.: automata theory and formal languages, graph theory, set theory, fixpoint theory, and the like. More advanced sub-topics of compiler construction (e.g.: garbage collection, parallelism, automatic type inference, compilation of object-oriented source languages with classes and sub-classes, operator overloading, dynamic typing, dynamic scoping, and the like) could merely be mentioned cursorily within the above-mentioned organisational limits of this introductory course. Nonetheless, to keep in touch with the most recent developments in the field, also a short overview-essay about obfuscating compilers (a.k.a. 'crypto-compilers') had to be written, whereby I provided the students with some initial literature references from which they had to start with their reading-and-writing work. Five working days were allocated to this essay task.

W.r.t. the related work recapitulated above, I introduced a self-made imperative procedural Student's Programming Language (SPL) the context-free (albeit initially ambiguous) grammar of which contained merely the following lexical and syntactic concepts:

- a main program, optionally followed by procedure definitions (sub-routines);
- simple input/output commands;
- named variables with preceding type declarations;
- imperative assignment statements with some simple arithmetic operations on composite terms;
- conditional statements with composite condition-terms (and optional else);
- unbound iteration statements (while with composite exit condition terms);
- bound iteration statements (for, with non-composite exit conditions and loop-counter increment +1);
- call statements for the parameter-less sub-routines.

Further details of the grammar need not be provided in this short report, as every competent computer science lecturer can easily design a similar SPL. In order not to overwhelm the students with programming work, the SPL sub-routines were truly old-fashioned without any input parameters and without any return-values. Proper function calls with input parameters and return values on a runtime stack were only discussed in the lectures on the basis of the book [17]. Though SPL sub-routines can contain inner (non-global) variables in separate static-semantic scopes, their main purpose is the manipulation of global variables by way of side-effects. Even the nesting of inner sub-procedures (with their own static-semantic scopes) within procedures was a grammatical possibility.

Following the classical phases of compiler construction, the students first had to come up with regular expressions (RE), then non-deterministic finite automata (NFA), then deterministic finite automata (DFA), finally minimised finite automata (MFA) for the vocabulary of SPL. For the above-mentioned pedagogical purposes no lexer-generator was allowed to be used; the students had to

implement their own lexers from their MFA although the availability of lexer-generators for professional purposes was mentioned in the lectures. In the next phase of the project, the deliberately ambiguous grammar of SPL had to be made non-ambiguous, and the students' own parsers for it had to be demonstrated. For the sake of in-depth understanding by construction it was again forbidden to use already available parser generators, the existence of which for professional purposes was only mentioned in the lectures. Due to the detailed instructions in the chosen book [17] about how to build a parser, this sub-task of the project was well done by all students in the course. As usual, the emitted concrete syntax tree also had to be purified to a less dense abstract syntax tree (AST) in an after-phase of the parse procedure.

More challenging, however, was the implementation of the static semantic analysis software in the last analytical (front-end) phase of the project. The large amounts of AST 'crawling' with all the inherited and synthetic attributes needed for the identification of the (nested inner) name-scopes as well as for type-checking, value-flow-analysis (etc.) turned out to be problematic particularly for those students who were weak in algorithms and data structures (study-year 2), because the chosen compiler construction book did not go deeply enough into the details of these matters; for pedagogical reasons (*higher* education) the students had to seek, find, and self-study whatever literature they could need for this phase. Hence, in several of the software demonstration sessions, types were not always correctly checked, or name-scopes were not consistently separated from each other. Especially in this static-semantics phase of the practical projects, several students have obtained more insight from their mistakes and errors than from their positive achievements.

After the static-semantic analysis, the generative (back-end) phase of the project ended with the production (and subsequent optimisation on the basis of variable liveness and variable dependency analysis) of intermediate code; the principles of producing hardware-specific target code from hardware-independent intermediate code were merely lectured along the lines of [17]. This last phase of the practical project was not too difficult for the students as the algorithmic generation of intermediate code (from AST and static-semantic information) was well described in the chosen book.

The ancient programming language BASIC in its simplest non-modernised form was chosen as the target language for the students' intermediate code generators. In its oldest form, BASIC is a pure *von-Neumann language* with its notoriously "harmful"[8] GOTO jumps to symbolic addresses. Thus all the high-level control structures (if-then-else, while, for) with their composite logical branching conditions had to be translated by the students' code generators into cascades of GOTO jumps, as it would also have been the case in genuine machine code (for specific hardware) at the very end of the code generation chain. The automatically generated BASIC programs were then further optimised by some of the not-so-difficult techniques which the chosen book described in sufficient detail (e.g.: common sub-expression elimination, constant propagation, and the like). Run-time tests with carefully chosen input values were used to quick-check

whether the students' implementations of 'optimisations' had actually damaged the operational semantics of the un-optimised BASIC programs — in several demo sessions that was indeed the case. After looking at their thus-generated BASIC code, many students were astonished about its mind-boggling cascades of GOTO jumps. Thus the students also began to appreciate the concern of Dijkstra's famous 'GOTO harmful' letter [8] (the reading of which was an additional homework task), and began to understand that it is now the compiler's function to create those low-level GOTO jumps which the human programmer is no longer supposed to write. Because BASIC emulators are nowadays available on the Internet, the students could use those emulators to see with their own eyes how their own SPL input programs could be translated (if free of lexical, syntactic and static-semantic defects) by their own compilers to executable BASIC code, and how a subsequent run would proceed step by step in the observable BASIC emulators.[2] Insofar as the elements of SPL can be easily expressed in familiar languages like JAVA, for which well-tested compilers are already available, the students were thus also able to conduct further comparative experiments and observations by first re-writing an SPL program to JAVA and then seeing whether the BASIC behaviour of their compiled SPL program would match the runtime behaviour of the corresponding JAVA program. If thus, for example, a student's scope checker in the static-semantic analysis phase would still contain some undetected flaw, then his finally generated BASIC program could be expected to reveal in the online-emulator a strangely different runtime behaviour in comparison against the runtime behaviour of the SPL-equivalent JAVA program after its translation to byte code by a trustworthy JAVA compiler.

All those features provided the students with fruitful study-experiences, and the pass rate after the course's final exam was remarkably high. Anecdotal evidence (from students' various e-mails) seems to indicate that the students have by-and-large appreciated my educational approach as well as the value of the knowledge obtained from it. One student remarked explicitly that he now grasps why I had called compiler construction *computer science in a nutshell* at the very beginning of the course.

3 Possible Critique and Outlook to Future Developments

In some internal discussions with several colleagues before I wrote this paper a number of interesting questions had been asked — for example: why was it not allowed to use already existing lexer and parser generators? Would the use of such pre-existing tools not provide the students with the same insight as the tedious creation of their own lexers and parsers? Indeed there are some 'practically oriented' books like [20] which do not delve as deeply into the underlying theoretical concepts as we did, and I am also aware of at least one university in the country where the topic is presented in such an overview-oriented style. If, however, we are willing to accept the epistemological opinion that an engineer can

[2] As a minor by-product of this approach, the students also obtained some insight into the *history of programming languages* and computing.

fully grasp only what he can construct, then the students' own lexers and parsers (no matter how simplistic) will be of better engineering-educational value than the mere inspection of the software code of already existing lexer- and parser-generators. By analogy: it is also not sufficient to inspect a car to become an automotive engineer. Why was the simple SPL, rather than a modern language like Python, used as source language? Here my answer is: for the students in the limited above-mentioned set-up of my course, the problems of type-checking and code-generation for object-oriented source languages would have been too difficult. Object-oriented type-checking alone (let alone code-generation) would have required a theoretical foundation along the lines of [6] which would have been far outside the scope of our curriculum. As SPL in all its simplicity is already Turing-complete it sufficed for the implementation of the usual pedagogical example programs (like Euclid's GCD algorithm) from which the students were able to generate executable target code with reasonable effort. Why was the old BASIC, and not a modern language like Python used for target code? My answer is that BASIC is a proper von-Neumann language, with sybmolic addresses (line numbers) and GOTO jumps like in genuine machine code, which is at the same time observably executable in a number of freely available interpreters and emulators. Python, with all its high-level features, is *no* such von-Neumann language and is thus far away from resembling machine code in any form. With the available BASIC interpreters the students were able to empirically observe the runs of their self-translated SPL programs in the BASIC interpreter line by line. For all the above-mentioned reasons I hope that this short experience report can serve both as a motivation and as a recipe for the (re)introduction of compiler construction at other institutions of higher education anywhere in the world. With the recent growth in new high-level special-purpose-languages (like specification- or modelling languages for software engineers), or the recent emergence of crypo-computers for which code-obfuscating compilers (a.k.a. crypto-compilers) are needed, the topic of compiler construction might soon get rid of its (prejudiced) smell of oldfashioned-ness and might come back into the centre even of IT-commercial interests.

Acknowledgements. Thanks to my students who provided comments on the educational quality of my compiler construction course. Thanks to *B. Watson, D. Watson* [20], as well as *T. Mogensen* [17] for interesting conversations on this topic. Thanks also to *P. Breuer* for his valuable hints to the growing research on crypto-compilers [5]. Last but not least thanks to the *anonymous reviewers* of SACLA'2019 for their critical and helpful comments on the draft of this short-paper prior to its publication.

References

1. Aho, A.V.: Teaching the Compilers Course. ACM SIGCSE Bull. **40**(4), 6-8 (2008)
2. Aiken, A.: Cool: a Portable Project for Teaching Compiler Construction. ACM SIGPLAN Not. **31**(7), 19-24 (1996)

3. Appelbe, B.: Teaching Compiler Development. In: Proc. SIGCSE'79 10th ACM SIGCSE Techn. Symp. on Comp. Sc. Educ., pp. 23-27 (1979)
4. Baldwin, D.: A Compiler for Teaching about Compilers. In: Proc. SIGCSE'03 34th ACM SIGCSE Techn. Symp. on Comp. Sc. Educ., pp. 220-223 (2003)
5. Breuer, P.T.: Compiled Obfuscation for Data Structures in Encrypted Computing, arXiv:1902.06146; Compiling for Encrypted Computing: Obfuscation but Not in Name, arXiv:1902.06146, (2019)
6. Bruce, K.B.: Foundations of Object-Oriented Languages: Types and Semantics. MIT Press (2002)
7. Demaille, A., Levillain, R., Perrot, B.: A Set of Tools to Teach Compiler Construction. In: Proc. ITiCSE'08 13th Ann. ACM Conf. on Innov. and Techn. in Comp. Sc. Educ., pp. 68-72 (2008)
8. Dijkstra, E.W.: Go To Statement Considered Harmful. Comm. ACM **11**(3), 147-148 (1968)
9. Griswold, W.G.: Teaching Software Engineering in a Compiler Project Course. J. Educ. Resour. Comput. **2**(4), paper #3 (2002)
10. Gruner, S.: On the Future of Computer Science in South Africa: A Survey amongst Students at University. In: Proc. SACLA'15 44th Ann. Conf. of the Southern Afric. Comp. Lect. Assoc., pp. 215-219, (2015)
11. Henry, T.R.: Teaching Compiler Construction using a Domain Specific Language. In: Proc. SIGCSE'05 36th ACM SIGCSE Techn. Symp. on Comp. Sc. Educ., pp. 7-11 (2005)
12. Kruss, G., Visser, M.: Putting University-Industry Interaction into Perspective: A Differentiated Vew from Inside South African Universities. Journ. Technol. Transf. **42**(4), 884-908 (2017)
13. Maharasoa, M., Hay, D.: Higher Education and Graduate Employment in South Africa. Quality in Higher Educ. **7**(2), 139-147 (2001)
14. Mahoney, W., Pedersen, J.: Teaching Compiler Code Generation: Simpler is Better. ACM SIGCSE Bull. **41**(4), 30-34 (2010)
15. Martin, D.: An Emulator used to Teach Compiler Design. In: Proc. 15th Ann. ACM Southeast Regional Conf., pp. 1-10 (1977)
16. McKenna, S., Powell, P.: 'Only a Name Change': The Move from Technikon to University of Technology. Journ. Indep. Teaching and Learn. **4**(1), 37-48 (2009)
17. Mogensen, T.Æ: Introduction to Compiler Design. 2nd ed., Springer (2017)
18. Shapiro, H.D., Mickunas, M.D.: A New Approach to Teaching a First Course in Compiler Construction. In: Proc. SIGCSE'76 ACM SIGCSE-SIGCUE Techn. Symp. on Comp. Sc. Educ., pp. 158-166 (1976)
19. Vegdahl, R.: Using Visualization Tools to Teach Compiler Design. In: Proc. CCSC'00 14th Ann. Consortium on Small Colleges Southeastern Conf., pp. 72-83, Consortium for Comp. Sc. in Colleges (2000)
20. Watson, D.: A Practical Approach to Compiler Construction. Springer (2017)
21. White, E., Sen, R., Stewart, N.: Hide and Show: Using Real Compiler Code for Teaching. In: Proc. SIGCSE'05 36th ACM SIGCSE Techn. Symp. on Comp. Sc. Educ., pp. 12-16 (2005)
22. Xu, L., Martin, F.G.: Chirp on Crickets: Teaching Compilers using an Embedded Robot Controller. In: Proc. SIGCSE'06 37th ACM SIGCSE Techn. Symp. on Comp. Sc. Educ., pp. 82-86 (2006)