TEACHING LANGUAGE THEORY AND AUTOMATA: A COMPILER GENERATION ORIENTED APPROACH USING ATOCC

M. Hielscher, Chr. Wagenknecht

PHBern – University of Teacher Education
Centrum for Computer Education
Muesmattstrasse 29, 3012 Bern, Switzerland
mail@michael-hielscher.de

Hochschule Zittau/Görlitz
Fachbereich Informatik
Brückenstraße 1, 02826 Görlitz, Deutschland
c.wagenknecht@hs-zigr.de

Abstract

Teaching Language Theory and Automata (LTaA) in such a way that students are highly motivated to actively learn is quite a challenging task. This paper presents a pedagogical approach that connects suitable parts of these rather abstract topics with some applications in automated compiler construction, i.e. compiler generation. To get this approach implemented in a real class situation we have developed an appropriate learning environment, called AtoCC. To illustrate how AtoCC can be used to support teaching as well as learning, an extensive exercise on compiler generation, which the authors lecture on, is presented.

1. Introduction

Language Theory and Automata (LTaA) is an integral part of computer science studies at university level ([4]). "Practice makes perfect" expresses the fact that knowledge and methods must be trained in order to internalize. However, practicing the abstract contents of LTaA appears to be impossible or rather unattractive. Most courses at universities are based on series of theorem-proof-example-blocks with a very limited reference to practical importance. LTaA is therefore often felt to be boring and too theoretical . Thus students adopt a negative attitude to the subject even before thy take an LTaA course.

In order to overcome difficulties like that we were looking for motivating practical applications of the theoretical contents that have to be taught. Compiler construction (CC) as a part of the practical computer science component meets our needs. It applies both, knowledge about formal languages and abstract automata, which is well proved by CC courses offered by many universities. In most of the cases such courses are built on the top of a long theory oriented section, which definitely does not improve the individual motivation of the students sitting in a theory class.

A good mix of theory and practice is often referred to as the secret of success. With regard to LTaA, CC can serve as a source of motivation. Practical needs and necessary theoretical knowledge should mutually facilitate and require each other.

For motivating students for the LTaA lecture, we define the goal of developing our own compiler till the end of the course. To solve this extensive task we split it into smaller parts like: what is a language and syntax, how we can check if a word belongs to a given language, how a compiler works (in general) and how individual parts like scanner and parser work. In each lecture the students practically develop necessary parts for the final compiler project in a theory-apply-cycle. This can be referred to as a theory on demand process.

According to the pedagogical concept briefly described above we could force the students to create their own compiler by programming a scanner and parser by hand utilizing their favorite programming language. In fact, this is a challenging programming task; however, students will not learn really much about the concepts of LTaA and how to apply them on various problems aside from the concrete example we have chosen for the lectures. It seems more promising to teach the concepts of LTaA and apply them to automated compiler generation. CC in terms of compiler generation means to apply a generator to appropriate descriptions of both, the source language and the target one. It is fundamentally different from perceiving compiler construction as a craftsmanship.

The well known CC tools like LEX and YACC (FLEX and BISON) are made for expert engineers and not for educational purposes. There are a lot of technical pitfalls that can be time consuming to deal with. Tools like that are not designed for the hands of students.

We use the computer-based learning environment AtoCC ([1]) which was especially built to support LTaA lectures based on the concept introduced here. The outlined pedagogical approach is illustrated in more detail in section 2 along with an example used in a real class situation. The results and their evaluation are summarized in section 3. The early observations are promising,but still not sufficient to make a quantitative analysis yet.

2. Our course

It is significant to choose motivating examples. Therefore it is highly recommended to avoid choosing very simplified languages like a^nb^n. To select assembler and machine code to be the target language of a compiler project is also not a good choice. From a student's perspective it is more interesting to translate their own language into a visual (e.g.: SVG) or acoustic (e.g.: MIDI) one. The source language should be complex enough to show a practical relevance as well as scalable enough to provide limitations on some representative language elements.

In this section the solving process of an extensive exercise is described. At this point the students have already worked with finite and push-down automata, regular expressions and formal grammars in previous exercises. However, we can use all components of AtoCC to reflect back on theory to be applied in this exercise.

The source language for this example is a robot language called Drawing Robot (DR). One can think of a drawing robot as a turtle well-known from the turtle geometry, being part of the logo

programming language. It is a moveable screen icon equipped with a pencil on the waist. While putting the pencil on the visual playground, a trace is drawn along with the robot's movement. The language is not too trivial because brackets are used for loops, so at least a push-down automaton is required to describe it.

2.1. The task as given to the students

Develop a compiler that translates the language DR into PDF. The following sample program in DR produces the tiny rosette in fig. 1:

```
L 36 [L 4 [F 100 R 90] R 10]
```

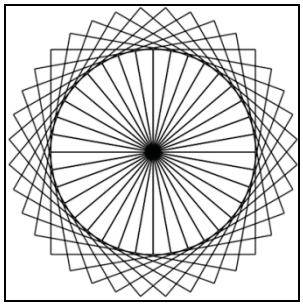

Figure 1: Sample output for a DR program

To write DR programs the following commands can be used:

```
F n           forward n steps
R n           turn right for n degrees
L n [ ... ]   loop n-times the content of the brackets
COLOR f       pen color f must be red, green, blue or black
PEN n         changes the stroke size of the pen
```

Hint: PDF itself is a binary format and cannot be easily described by human beings as the target language generated. That is why you should aim for something like a multi step strategy of the transformation process from DR to PDF. Look for suitable tools to simplify this compilation process.

2.2. Model the compilation process

The students have to model the process that takes a DR program and compiles it to a final PDF document which is considered as a program written in PDF language. According to the hint provided for the task described in the previous section, the students are not able to follow the idea of a direct translation from DR to PDF. In previous exercises the command line tool PS2PDF (from GhostScript) was used to produce PDF from PostScript (PS) files. PS is a language readable for human beings and therefore it can be used as the target language for a DR to PS compiler. We have a two-step compilation process from DR to PS and then from PS to PDF. We can visualize and model this process with the help of T-diagrams (first introduced in [3]).

T-diagrams can be developed easily with pen and paper. In fact, this method makes it difficult for students to accomplish the connection rules defined for the shapes to produce a valid diagram. Instead of paper the students use TDiag providing highlighting and validation support. Furthermore, TDiag allows us to automatically execute a diagram bonded to real files on the local system. Invoking a diagram performs the execution of compilers and programs in the sequence defined trough the diagram. In fig. 2 a student's solution for the DR example is shown (concrete filenames are not included). The T-diagram obviously lacks the DR2PS compiler. This is exactly the piece of software that has to be developed by the students. Executing this diagram will later on result into:

```
java DR2PS input.dr output.ps
ps2pdf output.ps
foxitreader output.pdf
```

Beside of the DR to PS compiler all the other software components are already installed on the system the students are working on.

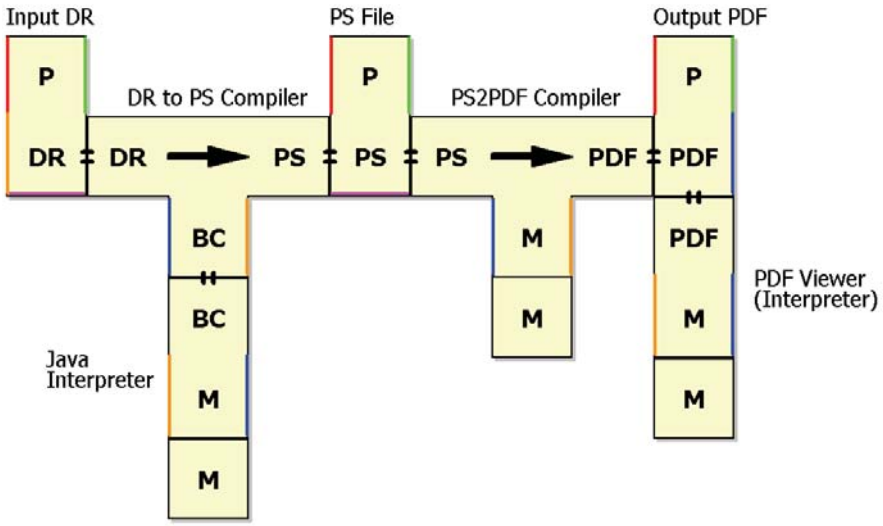

Figure 2: T- diagram for the DR to PDF translation

2.3. Defining DR

The DR to PS compiler involves two languages that need to be declared and discussed before creating the missing compiler. The students have to define a context free grammar for DR based on the task description and sample DR programs. The students can use kfG-Edit from AtoCC to create a valid grammar and derive the examples given in the task description. The student may use the manual derive option within kfG-Edit to identify mistakes. As a result the students define a context free grammar for DR like this (represented in BNF):

$$
\begin{aligned}
G &= (N, T, P, s) \text{ with:} \\
N &= \{Program,\ Statements,\ Statement,\ Colorvalue,\\
&\quad\ Number,\ Digits,\ Digit,\ firstDigit\} \\
T &= \{\texttt{L},\ \texttt{[},\ \texttt{]},\ \texttt{COLOR},\ \texttt{F},\ \texttt{PEN},\ \texttt{R},\ \texttt{black},\ \texttt{blue},\\
&\quad\ \texttt{green},\ \texttt{red},\ \texttt{0},\ \texttt{1},\ \texttt{2},\ \texttt{3},\ \texttt{4},\ \texttt{5},\ \texttt{6},\ \texttt{7},\ \texttt{8},\ \texttt{9}\} \\
P &= \{Program \rightarrow Statements \\
&\quad\ Statements \rightarrow Statement\ Statements \mid \varepsilon \\
&\quad\ Statement \rightarrow \texttt{L}\ Number\ \texttt{[}\ Statements\ \texttt{]} \\
&\quad\ \mid \texttt{F}\ Number \mid \texttt{R}\ Number \\
&\quad\ \mid \texttt{COLOR}\ Colorvalue \mid \texttt{PEN}\ Number \\
&\quad\ Colorvalue \rightarrow \texttt{black} \mid \texttt{blue} \mid \texttt{green} \mid \texttt{red} \\
&\quad\ Number \rightarrow firstDigit\ Digits \\
&\quad\ Digits \rightarrow Digit\ Digits \mid \varepsilon \\
&\quad\ Digit \rightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \\
&\quad\ firstDigit \rightarrow \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}\ \} \\
s &= Program
\end{aligned}
$$

2.4. A scanner and parser for DR

As one part of the final DR to PS compiler the students have to create a scanner which tokenizes DR programs. For this purpose they use VCC from AtoCC to define a scanner description with token classes and associated patterns (regular expressions). For the sublanguage of *Number* one may define a regular expression pattern like `[1-9][0-9]*`. This pattern can be evaluated and simulated on a random text in RegExpEdit. When using regular expressions (RegExp) it is always a good idea to generate a regular grammar and a finite automaton for the associated regular language. The students can compare the expected behavior with the one of the automaton in AutoEdit (see fig. 3).

The second part needed for the DR to PS compiler is defining a parser with translation rules. Once again we use VCC which offers developing a scanner and a parser in a single project file and seamlessly connect both parts together. The parser can be easily derived from the grammar *G* previously defined by the students. Some of the production rules of *G* can be reduced because we already accommodated them within the regular expressions of the scanner description (like *Number*). The remaining productions can be directly transferred to the parser definition in VCC (see fig. 4).

VCC currently supports Java, C#, Delphi and Scheme as programming languages the generated compiler program is implemented in. After having chosen one of them, the student can produce a DR to DR compiler written in the selected language without making any adjustments. Such a compiler will output an input DR program without any changes. However, the compiler already performs a complete syntax check and will only output valid DR programs. The students can now perform little adjustments to change this predefined output behavior.

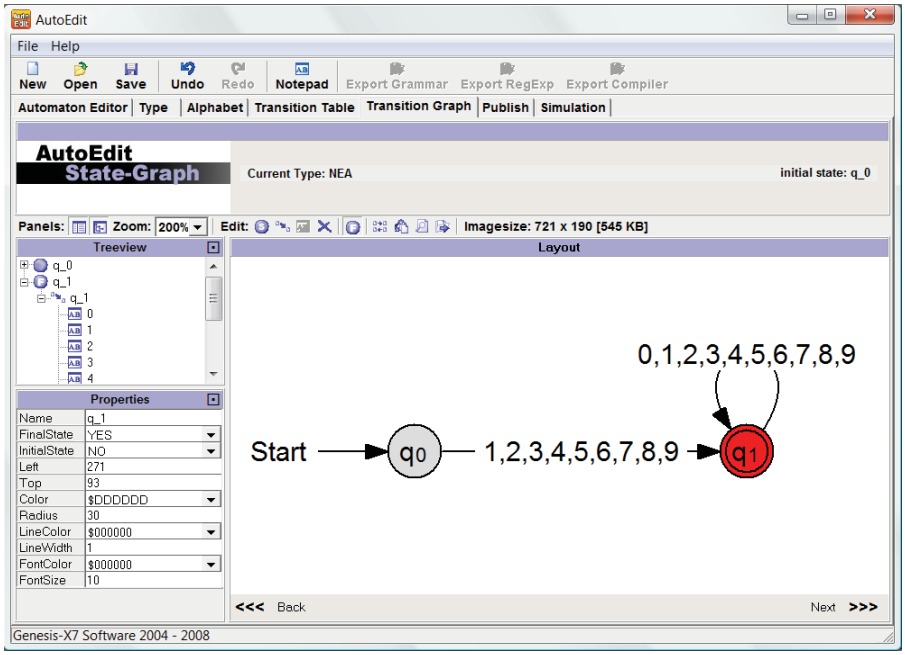

Figure 3: A finite automaton represented with AutoEdit

2.5. Adding the target language PS

In order to generate PostScript the students have to learn some basics of PS first. For the very limited operations of DR, only a few commands of PS are required. An appropriate worksheet is given to the students to help them.

The parser generated by VCC executes a small code fragment (a so called S-attribute) to produce its output whenever a parser rule is successfully applied. A java expression describing the value of an S-attribute for the production rule *Statement* → `COLOR` *Colorvalue* may look like this:

```
if ($2.equals("blue"))  $$ = "0 0 255 setrgbcolor ";
if ($2.equals("red"))   $$ = "255 0 0 setrgbcolor ";
if ($2.equals("green")) $$ = "0 255 0 setrgbcolor ";
if ($2.equals("black")) $$ = "0 0 0 setrgbcolor ";
```

The variable `$2` contains the literal value from the DR source program of *Colorvalue*. All elements on the right hand side of the production rule are numbered consecutively from `$1` to `$n`. The result of the rule *Statement* is stored in `$$`. A statement like `COLOR red` will therefore be translated into `255 0 0 setrgbcolor` (the command `setrgbcolor` is a PostScript method for setting the current pen color).

It is typical to add S-attribute expressions stepwise. After implementing a transformation rule, the students can preview their work by generating an executable compiler. Each rule which is not translated yet will just perform a pass-through based on the DR to DR compiler we started from.

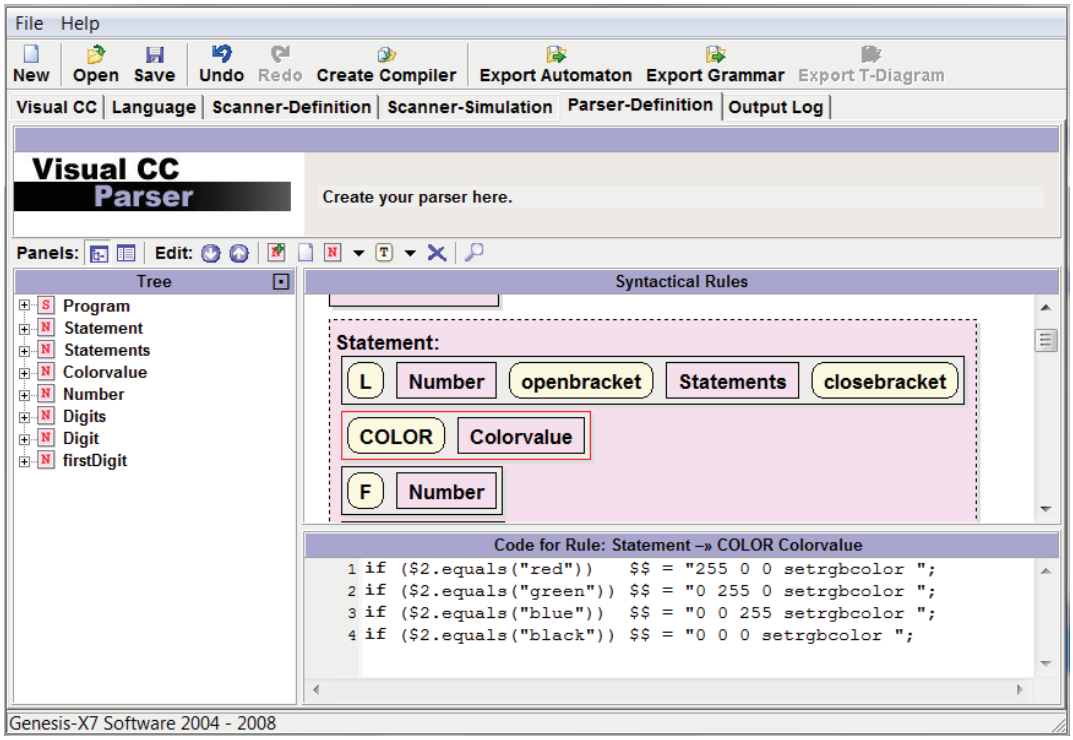

Figure 4: Parser definition represented in VCC

After having defined all translation rules the students let VCC generate the final DR to PS compiler. This compiler can be attached to the previous defined T-diagram within TDiag. From a pedagogical perspective it is very important to return to the level of modeling and validate this model by executing the diagram with real DR programs as input.

3. Results and evaluation

Computer science students from the University of Applied Sciences in Zittau/Goerlitz of the second and third semester attended LTaA classes were AtoCC was the only software used. In these classes the students successfully developed real compilers in tiny groups as a necessary requirement for their examination. The motivating topic was to develop a VCARD to SVG compiler (VCARD is a language for business cards). These languages are well described, but only minimal subsets of both were used. Enthusiastic students had a lot of possibilities to exceed the minimum requirements we had defined for this project. Some groups generated nicely colored business cards with background images and so on.

4. Conclusions

AtoCC supported our students to internalize and apply contents of LTaA. Especially context free grammars and their applications in automated compiler generation were highly motivating for the students. Some students even used AtoCC for their Bachelor/Master thesis projects with no or limited relations to LTaA topics, but where translation processes were involved. This lets us assume

that our pedagogical concept improves the students' ability to think in an abstract way by applying their knowledge of LTaA.

In numerous workshops facilitated by the authors, AtoCC was considered to be very useful for teacher education and even for use in secondary schools. The high quality export options in various data formats in almost all components of AtoCC, e.g. direct LaTeX output of automata tables, definitions and graphs, support teachers and students while producing learning material and project documentations.

Literature

[1] Hielscher M.: AtoCC Website, March 2009. http://www.atocc.de.
[2] Hielscher M., Wagenknecht C.: *AtoCC: learning environment for teaching theory of automata and formal languages*. In ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, page 306, New York, NY, USA, 2006. ACM.
[3] McKeeman W., Wortman D., Horning J.: *Compiler Generator (Automatic Computation)*. Prentice-Hall Englewood Cliffs, N.J., USA, 1970.
[4] T. J. T. F. on Computing Curricula: *Computing Curricula 2001 Computer Science, Final Report*. IEEE Computer Society, Association for Computing Machinery, 2001.
[5] Rodger S.: *Learning automata and formal languages interactively with jflap.* In ITICSE '06: Proceedings of the 11th annual SIGCSE conference on Innovation and technology in computer science education, page 360, New York, NY, USA, 2006. ACM.

Doručeno redakci: 6. 4. 2009
Recenzováno: 15. 6. 2009
Schváleno k publikování: 23. 6. 2009

FORMALE SPRACHEN UND ABSTRAKTE AUTOMATEN LEHREN: EIN ZUGANG VIA COMPILER-GENERIERUNG MIT ATOCC

Formale Sprachen und abstrakte Automaten (FSuA) so zu lehren, dass die Studierenden zu aktivem Lernen motiviert werden, ist eine echte Herausforderung. Dieser Aufsatz stellt einen didaktischen Weg vor, der geeignete Themen aus der theoretischen Informatik sinnvoll mit deren praktischer Anwendung im Compilerbau verknüpft. Um dieses Vorgehen umsetzen zu können, haben wir eine angepasste Lernumgebung (AtoCC) entwickelt. An einem konkreten Unterrichtsbeispiel wird vorgestellt, wie AtoCC sowohl den Lehrenden als auch den Lernenden unterstützt.

JĘZYKI FORMALNE A ABSTRAKCJA AUTOMATÓW NAUCZANIA: UDOSTĘPNIENIE DROGI TWORZENIA KOMPILATORA – GENEROWANIE Z ATOCC

Nauczanie języków formalnych i abstrakcyjnych automatów (FSuA) w taki sposób, aby stanowiło to dla studentów motywację do aktywnej nauki, jest prawdziwym wyzwaniem. Niniejszy artykuł przedstawia ukierunkowanie dydaktyki, łączące w sensowny sposób odpowiednie zagadnienia teorii informatyki z jej praktycznym zastosowaniem. W celu realizacji takiego podejścia opracowano narzędzie do nauki (AtoCC). Na konkretnym przykładzie zajęć przedstawiono, jak AtoCC wspomaga nauczycieli w nauczaniu i studentów w nauce.

FORMÁLNÍ JAZYKY A ABSTRAKTY AUTOMATICKÉ VÝUKY: ZPŘÍSTUPNĚNÍ CESTY VYTVÁŘENÍ KOMPILÁTORU – GENEROVÁNÍ S ATOCC

Vyučování formálním jazykům a abstraktním automatům tak, aby to motivovalo studenty k aktivnímu učení, je opravdovou výzvou. Tento příspěvek představuje didaktický směr, který smysluplně spojuje vhodná témata teoretické informatiky s jejím praktickým využitím. K realizaci tohoto postupu bylo vytvořeno vhodné studijní prostředí (AtoCC). Na konkrétním vyučovacím příkladu je znázorněno, jak je AtoCC vyučovací oporou pro učitele i studijní oporou pro studenty.