Visualizing Compiler Design Theory from Implementation Through an Interactive Tutoring Tool: Experiences and Results

Rafael Del Vado Vírseda[a]
Computing Systems and Computation Department, Complutense University of Madrid, Spain

Keywords: Compiler Design Course, Interactive Learning, Interactive Tutoring Systems, Compiler Writing Tools.

Abstract: In this paper, we analyze the experiences and results obtained by using an Interactive Tutoring Tool (ITT) (del Vado Vírseda, 2020; del Vado Vírseda, 2022) to interactively tutoring the learning of the basic theoretical contents of a course on compiler design. Instead of beginning by studying the theory and then obtaining the code of its corresponding implementation in each of the phases of compiler construction, we propose to start from the implementation obtained by the students using automatic code generation tools (del Vado Vírseda, 2021). By using ITT, we interactively guide the exploration of the finite state automata and graphs generated by compiler writing tools, learning the most important theoretical concepts from the implementation, and increasing the understanding of the theory in relation to the code of its implementation. This work reports on this educational experience of improving the teaching of theory from the implementation, by using the interactive visualizations and explorations produced by ITT. As an evaluation of the educational experience with ITT, the academic results obtained by the students are analyzed, to provide success indicators.

1 INTRODUCTION AND MOTIVATION

Although Compiler Design is one of the Computer Science (CS) subjects that best integrates theory and practice (Aho, 2008), there are many students' challenges in understanding Compiler Design theory. As we discussed in (del Vado Vírseda, 2021), a Compiler Design course usually focuses on realizing software projects in which our students are asked to develop a small compiler, or implement some variant of a compiler under study (Mak, 2009). The learning of the structure of a compiler is done from the implementation, focusing its design and implementation on the traditional form of a software engineering project (Waite, 2006). The main disadvantage of this classical approach is that overemphasizes students' programming skills. Students in a first Compiler Design course spending too much time on a software engineering portion of a compiler is not a problem (Nystrom, 2021), but classes covering compiler development also need to cover a lot of theoretical material. Thus, as we noted in (del Vado Vírseda, 2020), many students dedicate themselves exclusively to code development, often ignoring the theoretical concepts necessary for a correct implementation.

As a way of approaching this problem, alternative approaches to a Compiler Design course make use of compiler-writing tools (Demaille, 2008), ranging from `Flex/JFlex` and `Bison/CUP` to modern tools such as `ANTLR` or `LISA`, to help students write the implementations automatically. A compiler operates in phases, each on with its particularities, algorithms, techniques, and tricks. Students traditionally learn, in each phase in which a compiler is structured, a set of theoretical concepts about Compiler Design theory (lexical and syntactic analysis, syntax-directed translation, symbol table, type checking, code generation, etc.), and automatically implement these parts of increasing complexity with the tools before proceeding to the following phase. However, this educational approach still gives rise to a problem, as the tools often do not show students many of the details of their use that are related to theory. Therefore, in many cases, the tools do not serve to design a complete course whose objective is to obtain a theoretical understanding of how they work. This not only results in a large gap between practice and theory, but also results in students having an incomplete view of the course. As a result, students spend less and less effort and time studying their theoretical content.

In this paper, we start from the methodology de-

[a] https://orcid.org/0000-0002-1942-751X

DelVado Vírseda, R.
Visualizing Compiler Design Theory from Implementation Through an Interactive Tutoring Tool: Experiences and Results.
DOI: 10.5220/0011709800003470
In *Proceedings of the 15th International Conference on Computer Supported Education (CSEDU 2023) - Volume 2*, pages 333-340
ISBN: 978-989-758-641-5; ISSN: 2184-5026

scribed in (del Vado Vírseda, 2021) to increase the understanding of the theory of a Compiler Design course from the concrete implementation generated by the automatic code construction tools, according to the various stages that take place during the construction of a compiler. To achieve this goal, we have developed in this paper an Interactive Tutoring Tool (ITT), from the initial design ideas set out in our previous work (del Vado Vírseda, 2022) as a particular instance of an Interactive Tutoring System (ITS) (del Vado Vírseda, 2020). Unlike other educational tools for teaching compiler constructions (Boyer and Chitsaz, 2004; Henriques and et al., 2005; Chakraborty and et al., 2013), and following (del Vado Vírseda, 2020), each teacher can choose through ITT the compiler-writing tools that he/she deems most suitable for teaching the course, and combine them in a modular way in the same educational environment. If in the future you choose to change these tools or use another programming language, ITT will still be useful, since the interactive tutoring process performed by the ITT tool will continue to be valid. Moreover, the interactive tutoring process carried out by the ITT tool to explore the finite state automata and graphs generated by the compiler-writing tools w.r.t. to the code will remain analogous for tutoring students from implementation to theory.

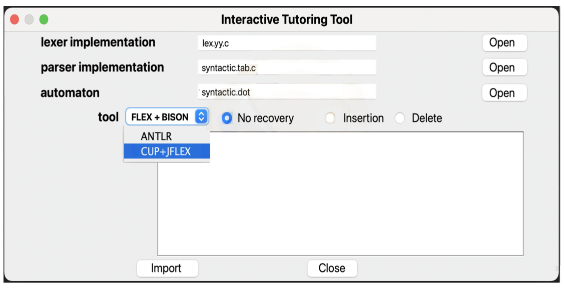
The paper is structured as follows. In the following section, we discuss the contributions to related work. Next, we describe the design and use of the educational tool ITT. We analyze the methodology and evaluation applied to obtain success indicators from our experiences and results. Finally, conclusions and an outlook for future work close the paper.

2 CONTRIBUTIONS AND RELATED WORK

Over the past two decades, several strategies have been designed to be applied in a course on Compiler Design within the CS curriculum (Waite, 2006). First, "emphasize design patterns, teamwork, and programming methodology by constructing a compiler to meet assigned specifications" (software project (Mak, 2009; Nystrom, 2021)). Second, "emphasize the role of theory to enable automation of compiler tasks, and illustrate the limitations of that theory" (application of theory (Aho et al., 2006)). Third, "emphasize the broad applicability of compiler technology to implement languages for special purposes" (computer communication support (Henry, 2005)). CS students tend to achieve higher motivation if they start as early as possible with a simple, self-built implementation that enables them to apply, as quickly as possible, each theoretical concept they acquire during their learning process, i.e., if they have something concrete that they can manipulate. For this reason, our work focuses on the second strategy under a different perspective, focusing on the function of the implementation obtained by using tools for the automation of the compiler tasks, in order to emphasize the role of theory. By this novel approach, we ensure that the importance of the software engineering strategy in a Compiler Design course is not underestimated, while avoiding overemphasis on parsing and syntax-directed translation theory. The connection with the third strategy is achieved through the use of ITT, which support computer communication through interactive sessions.

Following (de Oliveira Guimarães, 2007; Frens and Meneely, 2006; del Vado Vírseda, 2021; del Vado Vírseda, 2022), we have developed a different course content in which theory is presented progressively through various implementations of varying complexity. The first implementation is just a lexical-syntactic analyzer of a simple language of arithmetic (see Sections 3 and 4), and the last one is a compiler for a PL/0 language. The theoretical concepts are introduced incrementally through interactive sessions from the implementations, and are used as a motivation to understand the theoretical reasons that make the implementation works. In this way, the implementations are a motivation to study the theory of the course. Unlike other works (Mernik and Zumer, 2003; Henriques and et al., 2005), theory does not come before or after, but at the same time as the understanding of the implementation and its execution.

While most recent efforts to improve students' learning in Compiler Design have focused on designing new tools (Sondag et al., 2010; Henriques and et al., 2005; Chakraborty and et al., 2013), comparatively little research has focused on automating examinations to improve interactive students' learning. Our educational experience with ITT provides new evidence that switching the assessment of students' understanding of the theory from implementation through interactive sessions improve students' results. Following (Lorenzo, 2011), our approach presents the new ITT tool for the automatic evaluation of the theoretical and practical parts of a Compiler Design course. Moreover, in recent years, the number of papers including studies on the educational effectiveness of the diagnostic messages, produced by interpreters and compilers, has increased (Pettit, 2017). The ITT tool also uses compiler error messages as a pedagogical tool and as a part of the learning process of the Compiler Design theory. Taking (Becker, 2019) as a starting point, we have designed interactive learn-



Interactive Tutoring Tool

lexer implementation: lex.yy.c [Open]

parser implementation: syntactic.tab.c [Open]

automaton: syntactic.dot [Open]

tool: FLEX + BISON (ANTLR, CUP+JFLEX) — [✓ No recovery] Insertion Delete

[Import] [Close]

Figure 1: ITT Interface requesting implementation files.

ing sessions on compiler warnings, runtime errors, error messages, *shift/shift* and *shift/reduce* conflicts (Aho et al., 2006), to correct students' performance, track their progress, and tutor their study plans.

Finally, in the last two decades, a series of visualization tools have been designed to teach Compiler Design (Urquiza-Fuentes et al., 2011; Sondag et al., 2010). Visualization has been integrated frequently in Computer Science Education (Naps and et al.l, 2002), and these tools are very effective in helping students understand the transformation process from source programs at various stages of the compilation process (Vegdahl, 2000; Godbolt, 2021). For this reason, the interactive tool ITT allows the implementation to visualize more closely each theoretical concept in relation to its corresponding stage of the compilation process. Integrating these visualizations with the program's execution, ITT allows the student to improve the error correction and debugging with each of the levels of representation of the source language.

However, the ITT tool is not just another visualization tool for understanding code generation, but allows the interactive exploration of the finite state automata and graphs generated by automatic code generation tools underlying the implementation. The ITT tool enables the materialization of the theory from the implementation so that the student can manipulate them and better understand their relationship with the obtained code by compiler writing tools. Focusing on the interactive exploration of each automata portion of compilation generated by the compiler writing tools (as opposed to the full compiler tool chain), to interactively explain compiler theory and implementation instead of only automata visualization, is the primary contribution of this work.

3 THE INTERACTIVE TUTORING TOOL ITT

Our educational experience uses ITT as a particular instance of our previous interactive tutoring sys-

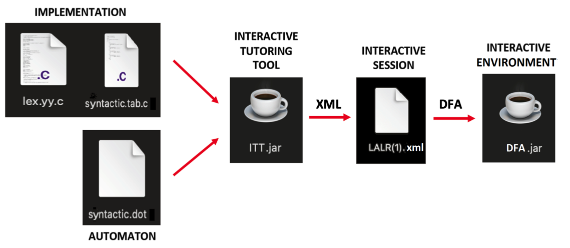


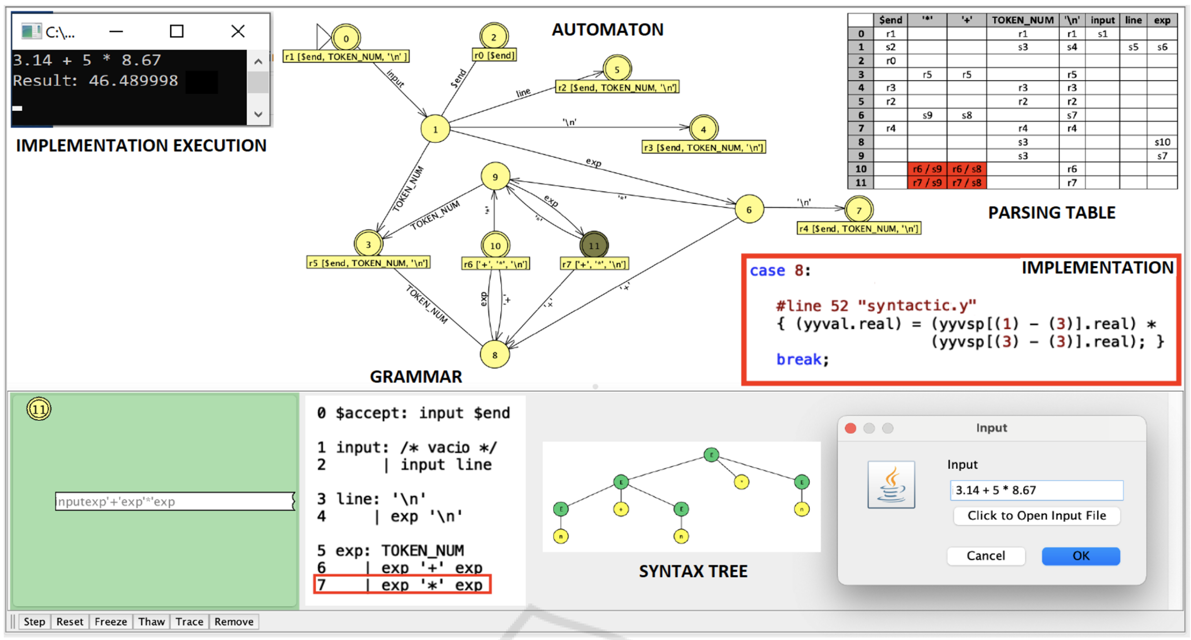
Figure 2: From the implementation to theory: Design and use of ITT for the interactive exploration of a DFA in XML.

tem (del Vado Vírseda, 2020), a computer system designed to interactively tutor traditional learning from theory to implementation in a Compiler Design course, providing immediate and customized instruction and feedback to our students. Unlike other similar educational systems (Mernik and Zumer, 2003; Henriques and et al., 2005), ITT can be used parametrically and modularly with a number of compiler-writing tools that the teacher considers most appropriate for the development of the course, following our methodology described in (del Vado Vírseda, 2021). Now, the main novelty is that with ITT students start now with their own implementations in `C/C++` or `Java`, along with DOT format files to generate and interactively explore the attribute graph and the finite automata underlying the code of the implementation, developing our preliminary ideas presented in (del Vado Vírseda, 2022). ITT interactively guides the students in the use of some theoretical tool, linking the most significant fragments of your code with the most classical theory concepts (del Vado Vírseda, 2021). We describe the use and experience of ITT through an example of one of the first grammars that books use to teach compilers (Aho et al., 2006).

Students begin their learning from a specification, provided by the teacher, of a very simple formal language, for example, for the recognition and evaluation of arithmetic expressions (Aho et al., 2006). This specification is composed, in the first place, by the file `lexical.l`, which contains the lexical specification of the language consisting of an intuitive regular expression to recognize (`yytext`) and evaluate numbers as positive integer and fixed-point values, together with zero (`yylval.real`) as `TOKEN_NUM` tokens:

```
  DIGIT [0-9]
{DIGIT}+('.'{DIGIT}+)?   { yylval.real = atof(yytext);
                           return TOKEN_NUM; }
```

Secondly, the student is provided with the syntax-semantics specification of the language through the file `syntactic.y`. This specification is formed by an intuitive attribute grammar, with which the student can introduce arithmetic expressions in different lines, each of them ending in a line break `'\n'`. In



```
3.14 + 5 * 8.67
Result: 46.489998
```

	$end	'*'	'+'	TOKEN_NUM	'\n'	input	line	exp
0	r1			r1	r1	s1		
1	s2			s3	s4		s5	s6
2	r0							
3		r5	r5		r5			
4	r3			r3	r3			
5	r2			r2	r2			
6		s9	s8		s7			
7	r4			r4	r4			
8				s3				s10
9				s3				s7
10		r6 / s9	r6 / s8		r6			
11		r7 / s9	r7 / s8		r7			

```
case 8:

  #line 52 "syntactic.y"
  { (yyval.real) = (yyvsp[(1) - (3)].real) *
                   (yyvsp[(3) - (3)].real); }
  break;
```

```
0 $accept: input $end

1 input: /* vacio */
2      | input line

3 line: '\n'
4     | exp '\n'

5 exp: TOKEN_NUM
6    | exp '+' exp
7    | exp '*' exp
```

Figure 3: Graphical and interactive ITT environment (1).

addition, the attributes and semantic equations necessary for the evaluation of its result are included:

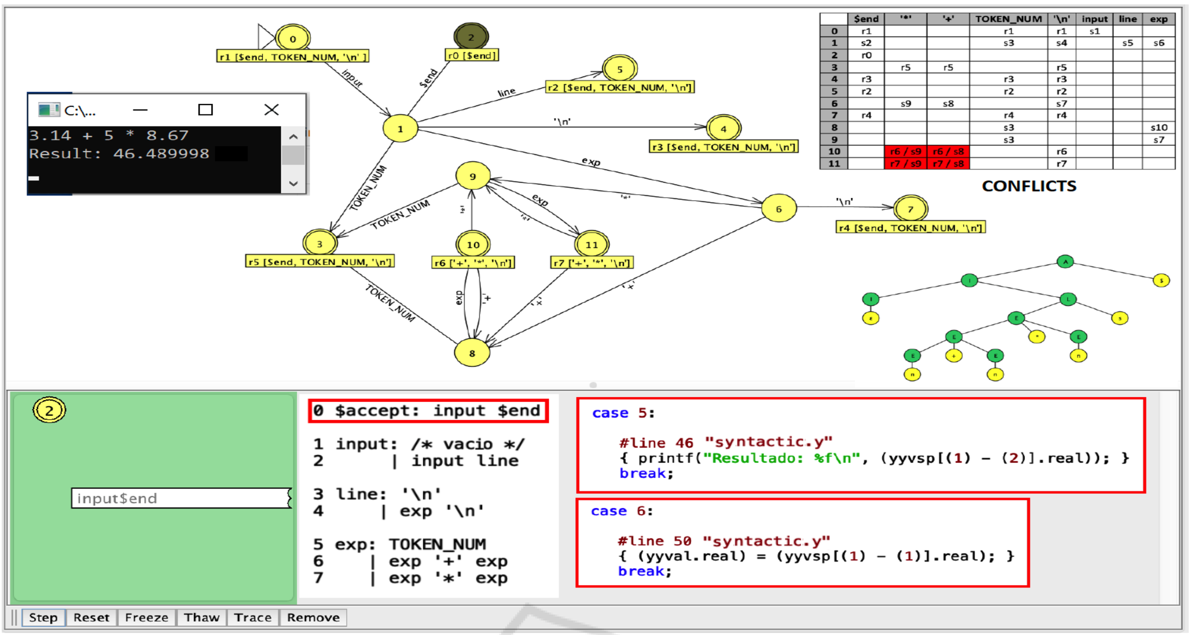
```
input :                                        /* r1
      | input line                             /* r2
      ;
line  : '\n'                                   /* r3
      | exp '\n' { printf('Result: %f', $1); } /* r4
      ;

exp   : TOKEN_NUM          { $$ = $1;      }   /* r5
      | exp '+' exp        { $$ = $1 + $3; }   /* r6
      | exp '*' exp        { $$ = $1 * $3; }   /* r7
      ;
```

From these two files, students generate their own implementation in C++ by using the Flex/Bison tools (Levine, 2009; del Vado Vírseda, 2022): The file lex.yy.c with the implementation of the scanner, and the file syntactic.tab.c with the implementation of the LR parser. In addition, the student gets the file syntactic.dot with the Deterministic Finite Automaton (DFA) underlying the implementation files. When the student executes ITT, implemented in a ITT.jar file, the student is asked to enter the files with the lexical, syntax-semantics implementation, the DFA automaton, as well as to indicate the concrete tools used to obtain the implementation, in this case, FLEX+BISON (see Figure 1). From these three files, ITT has all the information needed to create an interactive session, similar to other graphical tools as JFLAP (Rodger et al., 2011) (see Figure 2). The ITT tool translates the syntactic.dot file into the file LALR(1).xml in XML (Adams and Trefftz, 2004), so that it can be executed by our interactive environment DFA.jar, which contains the DFA associated to the grammar rules r1, r2, etc., and the links to the code fragments corresponding to its implementation in syntactic.tab.c:

```
<xml version="1.0" encoding="UTF-8" standalone="no">
   <structure>
        <type> dfa </type>
        <automaton>
            <!--The list of states.-->
            <state id="0" name="0">
                <x> 350.0 </x>
                <y> 49.0 </y>
                <label> r1 [$default_r1_link] </label>
                <initial/>
                <final/>
            </state>
            ............
            <!--The list of transitions.-->
              <transition>
                <from> 0 </from>
                <to> 1 </to>
                <read> input [$default_sift_link_01] </read>
              </transition>
              ............
        </automaton>
   </structure>
```

4 FROM IMPLEMENTATION TO THEORY

When the interactive session starts, ITT.jar opens a window in which executes the files with the im-



```
3.14 + 5 * 8.67
Result: 46.489998
```

	$end	'*'	'+'	TOKEN_NUM	'\n'	input	line	exp
0	r1			r1	r1	s1		
1	s2			s3	s4		s5	s6
2	r0							
3		r5	r5		r5			
4	r3			r3	r3			
5	r2			r2	r2			
6		s9	s8		s7			
7	r4			r4	r4			
8				s3				s10
9				s3				s7
10		r6 / s9	r6 / s8		r6			
11		r7 / s9	r7 / s8		r7			

```
0 $accept: input $end

1 input: /* vacio */
2       | input line

3 line: '\n'
4     | exp '\n'

5 exp: TOKEN_NUM
6    | exp '+' exp
7    | exp '*' exp
```

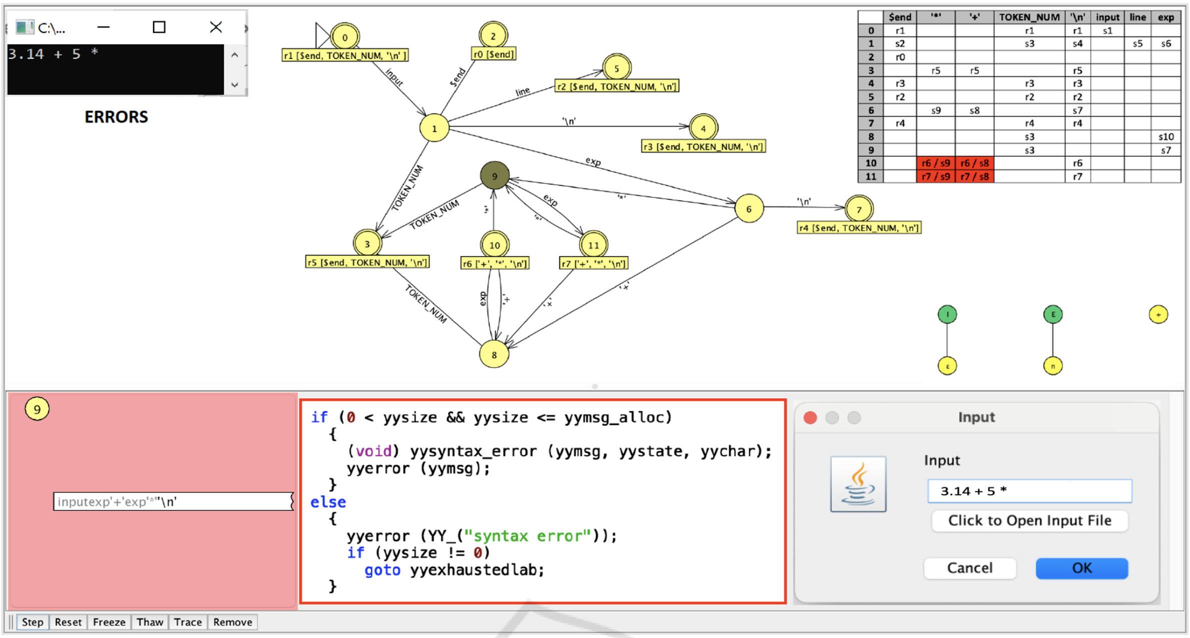
```
case 5:

  #line 46 "syntactic.y"
  { printf("Resultado: %f\n", (yyvsp[(1) - (2)].real)); }
  break;
```

```
case 6:

  #line 50 "syntactic.y"
  { (yyval.real) = (yyvsp[(1) - (1)].real); }
  break;
```

Figure 4: Graphical and interactive ITT environment (2).

plementation provided by the student. If the compilation is successful (errors are shown in the text box in Figure 1), the student enters an arithmetic expression as input, for example, $3.14 + 5 * 8.67$, and the result of the evaluation of the expression, in this case, 46.489998, appears in the window (see Figure 3, upper left corner). Next, `DFA.jar` opens a new window with the interactive session, in which the graphical visualization of the main theoretical contents underlying the implementation execution will appear (see Figure 3, upper part): The DFA responsible for directing the analysis process, and the state transition table with which to traverse the automaton from its initial state `0` to the different final states, highlighted with double circles. The student's goal is to enter sequences of tokens that represent and correspond to the arithmetic expression entered in the execution (e.g., students enter the sequence $3.14 + 5 * 8.67$, corresponding to the expression `inputexp'+'exp'*'exp'\n'`) as shown in the lower right corner of Figure 3. Thus, using the state transition table, the student will try to start from the initial state of the DFA and reach one of the final states, each of which has a grammar rule associated. When the student successfully reaches a final state (in the example, the student clicks on the final state `11`), ITT displays this fact in green color, and shows the student the grammar rule responsible for the reduction applied during the execution, in this case, the rule `r7` (i.e., `exp : exp '*' exp`). In addition, ITT shows the bottom-up construction of the syntax tree achieved up to that moment (see Figure 3, bottom), as well as the code fragment of the implementation in `syntactic.tab.c` for the reduction/shift, and the evaluation of the attributes in the semantic equation `$$ = $1 * $3` associated with the grammar rule `r7`. The student's ultimate goal is to interactively introduce sequences of tokens until successfully completing the construction of the syntax tree, as shown in Figure 4. During this analysis process, the student clicks on the final states to visualize and debug the complete code trace corresponding to the execution of the implementation, and at the same time, understand how the DFA works and how its associated state transition table is used. The student will also understand and identify the existence of DFA states in which *shift/shift* or *shift/reduce* conflicts (Aho et al., 2006) may appear, as in the example with states `10` and `11`, indicated in red in the table. Students can then perform several sessions with ITT to visually better understand why the value 46.489998 is obtained during the execution, instead of 70.5738, and why it is necessary to establish priorities and associativity between arithmetic operators to avoid ambiguity in the starting grammar given in `syntactic.y` (i.e., `%left '+'; %left '*'`).

Thus, ITT does not require the student to start with a correct grammar without conflicts and errors. The ITT tool offers functionality to debug grammars and resolve conflicts and errors, using them as an educational tool to deepen the understanding of theoretical design concepts (Becker, 2019). If at any time the



```
3.14 + 5 *
```

	$end	'*'	'+'	TOKEN_NUM	'\n'	input	line	exp
0	r1			r1	r1	s1		
1	s2			s3	s4		s5	s6
2	r0							
3		r5	r5		r5			
4	r3			r3	r3			
5	r2			r2	r2			
6		s9	s8		s7			
7	r4			r4	r4			
8				s3				s10
9				s3				s7
10		r6 / s9	r6 / s8		r6			
11		r7 / s9	r7 / s8		r7			

```
if (0 < yysize && yysize <= yymsg_alloc)
  {
    (void) yysyntax_error (yymsg, yystate, yychar);
    yyerror (yymsg);
  }
else
  {
    yyerror (YY_("syntax error"));
    if (yysize != 0)
      goto yyexhaustedlab;
  }
```

Input: 3.14 + 5 *

Figure 5: Interactive ITT debugging of errors and conflicts.

student enters an incorrect token sequence, or a final state with an incorrect reduction is reached, ITT displays it in red color (see Figure 5). In addition, ITT will show the part of the syntax tree it has built up to that point, as well as the code fragment of the implementation that is responsible for identifying this error. The student will better understand how to resolve these conflicts and errors, and if it is necessary, redesign the grammar in `syntactic.y` to obtain a new implementation with which to run again the interactive session provided by ITT.

5 METHODOLOGY AND EVALUATION

Following our previous methodology of evaluation in (del Vado Vírseda, 2021), Figure 6 shows the results achieved by means of the application of ITT by several groups of students in a Compiler Design course, over several consecutive academic years, on each phase in which a compiler is structured. The results of our experience have been obtained as the ITT tool and its underlying methodology have been developed and applied.

During the 2018-2019 course, ITT was tested for the first time with good results (see Figure 6), with 39 students contributing their ideas and criticisms for its development and implementation (17 with ITT and 22 in traditional course with only compiler writing tools). The average was 5.89 points, a step above the traditional method without ITT.

In the 2019-2020 course, ITT was applied for the first time in a controlled and supervised environment to evaluate its application (del Vado Vírseda, 2022). Students were carefully selected to follow the traditional tool-based learning method or using ITT throughout the course (to prevent students who are more likely to do well in the course are also more likely to choose to use ITT). Of the total number of students enrolled, 24 students were selected to use ITT and 17 students followed the traditional method. The students who followed the classical method obtained an average of 5.93 points, similar to the results already obtained in the 2018-2019 course. Students using ITT obtained now a higher average of 6.44.

The 2020-2021 academic year was affected by COVID-19, which resulted in classes going virtual. To facilitate the remote delivery of the main theoretical concepts, we developed in more detail the remote instance of ITT presented in Sections 3 and 4. On this occasion, 33 students were selected to follow the new learning method using ITT, and only 11 students were selected to follow the traditional learning method from theory to implementation, using notes and recorded lectures prepared by the teacher together usual compiler-writing tools. The average obtained by the students who used ITT increased to 7.0 points, while the average obtained by the students who followed the traditional method increased to 6.15 points. We have used specific quizzes for the topics where the ITT tool can be used, comparing the projects and

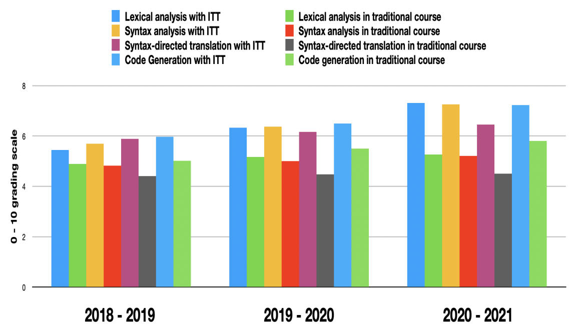
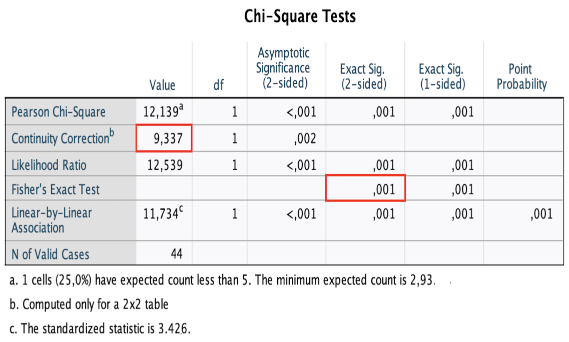


Figure 6: Results in each compiler phase with/without ITT.



Chi-Square Tests

	Value	df	Asymptotic Significance (2-sided)	Exact Sig. (2-sided)	Exact Sig. (1-sided)	Point Probability
Pearson Chi-Square	12,139[a]	1	<,001	,001	,001	
Continuity Correction[b]	9,337	1	,002			
Likelihood Ratio	12,539	1	<,001	,001	,001	
Fisher's Exact Test				,001	,001	
Linear-by-Linear Association	11,734[c]	1	<,001	,001	,001	,001
N of Valid Cases	44					

a. 1 cells (25,0%) have expected count less than 5. The minimum expected count is 2,93.

b. Computed only for a 2x2 table

c. The standardized statistic is 3.426.

Figure 7: Chi-Square tests of independence for the use and evaluation of the ITT tool in a Compiler Design course.

exam grades with and without using the tool (see Figure 6). The evaluation compares students learning under each approach in the same group, in which students following the traditional method only make use of the compiler-writing tools, instead of our tool ITT on the phases where the interactive tool can be used.

6 RESULTS AND SUCCESS INDICATORS

Finally, we have conducted statistical analysis to examine the effect of using ITT with students in a Compiler Design course. In the last year, the individual projects and exams have been the same, but there were two groups of students. Those who have followed the traditional method of learning from theory to the implementation provided by compiler-writing tools (without ITT), and those who have learned the theory of the course from the implementation with the help of ITT. We have tested, from the results and surveys presented in Section 5, the following research hypothesis: Students who learn the theory of a Compiler Design course from the implementation by using the interactive sessions generated by ITT, have been successful in the course by obtaining better results, motivation, and interested than with traditional learning methods without ITT.

For this purpose, we performed χ^2-tests of independence for hypothesis testing with the SPSS software and the calculation of confidence intervals of 95%. We tested H_0 (there is independence between the variables *ITT use* and *Success in the course*), and H_1 (there is no independence). The two variables are coded as follows: *ITT use* is coded with the number 1 if the ITT tool is used and 2 in the negative case, and *Success in the course* is coded with the number 1 if the final exam is passed and a positive motivation has been recorded, and 2 in the negative case. The statistical table in Figure 7 shows the value of the *continuity correction* statistic as a 2×2 table whose value was 9.337, with one degree of freedom. If we look at the *Fisher's exact* statistic for the *exact sig. (bilateral)* column, which we call p, this value indicates the probability of obtaining a difference between the groups greater than or equal to the observed one, under the null hypothesis H_0 of independence. As this probability is less than $\alpha = 0.05$ (since $p = 0.001 < 0.05$), we conclude with a significance level of 5% that the starting hypothesis H_0 should be rejected, so we must assume H_1 (i.e., the two variables are not independent, but associated). Therefore, the use of ITT to interactively learn theory from implementation tends to increase grades and motivation.

7 CONCLUSIONS AND FUTURE WORK

Compiler Design has resulted in a beautiful combination of practice and theory in Computer Science (Aho et al., 2006). However, theory and practice are not mutually exclusive, but are intimately connected, so that they coexist and support each other. Although new resources are being developed to balance theory and practice to get students better connect the practice and theory of compiler design (Nystrom, 2021), educational approaches based on traditional compiler books (Aho et al., 2006; Wirth, 1996) seem to have become obsolete to bridge the gap that students encounter between Compiler Design theory, and tools for designing modern compilers

This work has shown how to use the ITT tool to set interactive learning connections between the key phases in the implementations achieved by the compiler-writing tools, and specific portions of compiler theory. The use of ITT with an interactive graphical interface, to engage students and enhance the understanding of the theory from the implementation, gives students a first-hand understanding of how theory and practice can be beneficially interwoven.

Our main future work consists of enabling ITT to interactively use other tools, such as `LISA` or `ANTLRTree`, with which to better visualize the values of inherited and synthesized attributes from the code. We are also intending to use ITT with other compiler writing tools following (Chakraborty and et al., 2013). We are also working on using other implementation languages following (Godbolt, 2021), such as `C#`, `Java` code, provided by the `JFlex/CUP` tools, and `Python` implementations.

REFERENCES

Adams, D. R. and Trefftz, C. (2004). Using xml in a compiler course. *SIGCSE Bull.*, 36(3):4–6.

Aho, A. V. (2008). Teaching the compilers course. *SIGCSE Bull.*, 40(4):6–8.

Aho, A. V., Lam, M. S., Sethi, R., and Ullman, J. D. (2006). *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley.

Becker, B. A. e. a. (2019). Compiler error messages considered unhelpful: The landscape of text-based programming error message research. ITiCSE-WGR '19, page 177–210. ACM.

Boyer, T. P. and Chitsaz, M. (2004). Ice™ and ice/t™: Tools to assist in compiler design and implementation. *SIGCSE Bull.*, 36(4):55–57.

Chakraborty, P. and et al. (2013). A compiler-based toolkit to teach and learn finite automata. *Comput. Appl. Eng. Educ.*, 21(3):467–474.

de Oliveira Guimarães, J. (2007). Learning compiler construction by examples. *SIGCSE Bull.*, 39(4):70–74.

del Vado Vírseda, R. (2020). An interactive tutoring system for learning language processing and compiler design. In *ITiCSE'20*, page 552. ACM.

del Vado Vírseda, R. (2021). Learning compiler design: From the implementation to theory. In *ITiCSE'21*, pages 609–610. ACM.

del Vado Vírseda, R. (2022). ITT: an interactive tutoring tool to improve the learning and visualization of compiler design theory from implementation. In *SIGCSE'22*, page 1074. ACM.

Demaille, A. e. a. (2008). A set of tools to teach compiler construction. In *ITiCSE '08*, page 68–72.

Frens, J. D. and Meneely, A. (2006). Fifteen compilers in fifteen days. *SIGCSE Bull.*, 38(1):92–96.

Godbolt, M. (2021). Compiler explorer.

Henriques, P. R. and et al. (2005). Automatic generation of language-based tools using the LISA system. *IEE Proc. Softw.*, 152(2):54–69.

Henry, T. R. (2005). Teaching compiler construction using a domain specific language. *SIGCSE*, 37(1):7–11.

Levine, J. R. (2009). *flex and bison - Unix text processing tools*. O'Reilly.

Lorenzo, E. J. e. a. (2011). A proposal for automatic evaluation in a compiler construction course. In *ITiCSE '11*, page 308–312. ACM.

Mak, R. (2009). *Writing Compilers and Interpreters: A Software Engineering Approach*. Wiley Publishing, 3rd edition.

Mernik, M. and Zumer, V. (2003). An educational tool for teaching compiler construction. *IEEE Trans. Educ.*, 46(1):61–68.

Naps, T. L. and et al.l (2002). Exploring the role of visualization and engagement in computer science education. ACM.

Nystrom, R. (2021). *Crafting Interpreters*. Genever Benning.

Pettit, R. S. e. a. (2017). Do enhanced compiler error messages help students? results inconclusive. SIGCSE '17, page 465–470, New York, NY, USA. ACM.

Rodger, S. H., Qin, H., and Su, J. (2011). Changes to jflap to increase its use in courses. ITiCSE '11, page 339, New York, NY, USA. ACM.

Sondag, T., Pokorny, K. L., and Rajan, H. (2010). Frances: A tool for understanding code generation. SIGCSE '10, page 12–16. ACM.

Urquiza-Fuentes, J., Manso, F., Velázquez-Iturbide, J. A., and Rubio-Sánchez, M. (2011). Improving compilers education through symbol tables animations. ITiCSE '11, page 203–207. ACM.

Vegdahl, S. R. (2000). Using visualization tools to teach compiler design. volume 16, page 72–83, Evansville, IN, USA. Consortium for Computing Sciences in Colleges.

Waite, W. M. (2006). The compiler course in today's curriculum: Three strategies. *SIGCSE Bull.*, 38(1):87–91.

Wirth, N. (1996). *Compiler construction*. International computer science series. Addison-Wesley. slightly revised November 2005.