Improving Students’ Performance with Visualization of Error Recovery Strategies in Syntax Analysis

Jaime Urquiza-Fuentes, Francisco J. Almeida-Martínez, Antonio Pérez-Carrasco and J. Ángel Velázquez-Iturbide
Università Rey Juan Carlos de Madrid, Spain
Emails: {jaime.urquiza, francisco.almeida, antonio.perez.carrasco, angel.velazquez}@urjc.es

Visualization technologies can be used to enhance teaching materials. This work is focused on a complex aspect of syntax analysis, syntax error recovery strategies. Compiler visualization is not a new research field, but the authors have not found any visualization tool dealing with this topic. We have enhanced a parser visualization tool called VAST to produce educational visualizations regarding syntax error recovery strategies. We have evaluated the enhanced version of VAST and its visualizations about syntax error recovery strategies. We have detected that students who used the enhanced tool improved their performance developing syntax specifications with error recovery. Consequently, these students significantly improved their learning outcomes regarding error recovery exercises.


1. Introduction

Language processors and compilers are considered by students as one of the most difficult subjects in computer science degrees. The understanding of syntax analysis is mostly founded on the comprehension of the construction process of the syntax tree. The tree representation is quite close to the student’s mental representations of the syntax tree. Therefore its visualization can improve student’s understanding. Within syntax analysis, syntax error recovery strategies are one of the highly complex topics covered by these subjects. On the one hand, they require from students a deep understanding of the parsing techniques. On the other hand, their implementation within the parser generator tools is far from theory. For example, see comments about the panic mode strategy in a well-known book (Brown et al, 1992): “Proper placement of error tokens in a grammar is a black art ...”. We are interested in producing effective visualizations of syntax error recovery strategies. Our work is based on an existing parser visualization tool called VAST (Almeida-Martínez et al, 2009). We have enhanced this tool to produce visualizations of the syntax error recovery strategies.

The rest of the article is structured as follows. Next, we give a brief description of the parser visualization tool in Section 2. In Section 3 we describe the visualization of the syntax error recovery strategies, their design and implementation. We detail the evaluation process in Section 4. In
Section 5 we compare our work with the existing tools described in the literature. Finally, in Section 6 we state our conclusions.

2. Parser Visualization with VAST
VAST has been designed to cope with multiple parser generation tools and both parsing strategies, bottom-up (LR) and top-down (LL). In order to achieve parser-independence and keep ease of use as a fundamental requirement, VAST has been divided into two modules: VASTapi and VASTview. From a global point of view, VASTapi is devoted to producing data to be visualized while VASTview is devoted to producing visualizations and manage user interactions. The whole process must be adapted to the general parser development process, i.e. (1) writing the parser specification, (2) generating the parser source code with parser generator, (3) compiling the parser source code and (4) executing the parser application. Figure 1 shows a schematic view of the visualization process with VAST. First, the user has to provide a parser specification. This specification is automatically annotated with calls to VASTapi. Once the executable parser is generated, its execution produces, as a side effect, the source data for visualizations, stored in an XML file. Finally, VASTview produces the visualizations allowing the user to interact with them.

Figure 1: The parser visualization process with VAST.

2.1 Production of Visualization Data with VASTapi
The main information to visualize the syntax tree construction process is the application of grammar productions. VASTapi generates data that describe how grammar productions have been applied to build the syntax tree.

VAST accepts the specification languages used by the different generation tools, e.g. CUP\(^1\) (bottom-up) and ANTLR\(^2\) (top-down) are used in this article. Thus, students do not have to learn another different parser specification language. The parser specification is annotated with calls to VASTapi like addProduction ('Antecedent', 'Consecuent') for each grammar production. In order to associate the actual input stream with its parsing process, lexical specification is also annotated. This annotation consists in adding information to the tokens generated by the lexical analyzer: new mySymbol (yyline, yycolumn, token, VASTview_label, yytext()). Thus, the annotations


Improving Students' Performance with Visualization of Error Recovery Strategies in Syntax Analysis
are placed within the semantic actions area of both specifications, lexical and syntactical. The whole annotation process is automatically performed by VAST and transparent to the user.

The executable parser is the result of generating the parser source code, using standard parser generation tools, and compiling it. These simple steps are easily performed with VAST as well. The execution of the parser is as simple as providing the input stream and order VAST to execute the parsing process.

For each production applied during the parser execution, visualization data are generated. These data are stored in an XML file which contains the sequence of productions applied during the parser execution, i.e. the construction process of the syntax tree.

During the analysis of the input stream, there exist two alternatives: correct stream or syntax errors in the input stream. In both cases VASTapi generates an intermediate representation. However, in case of finding syntax errors, VASTapi allows to abort the analysis or to continue recovering from the previous error.

2.2 Visualizing the Parsing Process with VASTview

VASTview is the main component of the VAST user interface (see Figure 2) i.e., it is the module that produces the visualizations. It interprets the information stored in the XML file and produces the graphical representation that will be manipulated by the user. VASTview provides multiple views of the syntax analysis process. The syntax tree is displayed at the upper-left part of the interface. Views at the right and bottom offer additional information of the compilation process: the input stream, the parsing stack, the grammar and brief explanations of the actions performed on the syntax tree.

VASTview also manages user interaction with parser visualizations. On the one hand it includes a global view of the syntax tree to make easier users’ understanding of the syntax tree. On the other hand VASTview provides animation controls to visualize the syntax tree construction process synchronized with the corresponding stages of the input stream and the parsing stack.

![Figure 2: VASTview, the user interface of VAST](image-url)
The communication between VASTapi and VASTview is performed in an asynchronous way. Thus VASTview generates animations from the intermediate representation built by an annotated parser using VASTapi. To display each of the different views provided by VASTview some different information is needed. Here we describe how the different views are generated using VASTapi and VASTview.

The main view of VAST is the syntax tree view, enhanced with the global view. In this case VASTview requires information about the productions which have been applied, including the application order. VASTapi provides this information together with the links between the different productions. This task is performed during parser execution time.

The grammar specification provides the students with the grammar (using BNF notation) used in the syntax specification. This information is generated during parser execution time. When the annotated parser executes an action it remarks the applied production, thus the syntax tree construction process is easily interpreted.

The contents of input stream are provided by the user. The input stream view is synchronized with the syntax tree view, highlighting the last terminal symbol processed. This information depends on the processing order that is provided by VASTapi during parser execution time.

Finally, the stack and the actions explained views are generated during animation time. The nodes used to represent the syntax tree are the same nodes that will be pushed and popped from the stack. Graphical representations generated by VASTview are enhanced with textual explanations about the operations performed by the parser, e.g. production application.

3. Visualization of Syntax Error Recovery

Visualizing syntax error recovery should improve students’ understanding of this topic. VAST can be used with different parser generators; our students have used it with CUP (LR) and ANTLR (LL) parser generators. Here we describe how VAST visualizes the different error recovery strategies implemented in both parser generators. An early development can be found elsewhere (Almeida-Martínez et al., 2010).

Four syntax error recovery strategies are commonly described in the compiler books, e.g., (Aho et al., 2007): error productions, phrase level strategy, panic mode strategy and global recovery strategy. We have found that three of them are integrated in the parser generation tools, but little standardization exists in their actual implementations. Both tools, CUP and ANTLR, allow the use of error productions. CUP provides the panic mode strategy while ANTLR provides the phrase level strategy. Besides, ANTLR provides an own strategy mixing and modifying the standard ones. We do not visualize it because it is out of our educational objectives.

3.1 Visualization of Error Productions

Error productions detect concrete errors by specifying concrete grammar rules. This recovery is directly specified by the language designer (parser developer) using the errorProduction keyword in the parser specification. The error treatment is defined in the associated semantic actions of the error production, allowing it to perform a more specific error recovery process.

The implementation of the visualization of this strategy is simple. In order to display the application of error productions, VASTapi provides a new method called:

   addErrorProduction(‘antecedent’, ‘consecuent’)
VAST identifies the `errorProduction` keyword and annotates the associated rule with the previous method. The information communicated to VAST is exactly the same as in normal productions. VASTapi represents error productions highlighting them with a different colour. Figure 3 shows an example of this strategy. Here, the language designer implements the recovery process for typical errors where the programmer makes a mistake typing keywords, e.g., the “of” identifier instead of the “if” keyword.

![Figure 3: Visualization of the error production strategy: annotated grammar, input stream and graphical representation highlighting the error production with yellow colour](image)

### 3.2 Visualization of Phrase Level Error Recovery

Phrase level recovery is one of the simplest strategies. It tries to transform syntactically erroneous phrases into correct ones by inserting/deleting tokens in/from the input stream. Every state of a parser has a list of expected tokens that can be used in this strategy. A common example can be found in most C/C++ compilers with the insertion of the missing semicolon at the end of sentences. The implementation of the visualization uses some facilities from the ANTLR parser generator. ANTLR implements this method by inserting expected tokens into the stack. The parser developer can customize this procedure by overloading the method `displayRecognitionError`. This modification is specified in the parser specification, and can be automatically performed by VAST during the annotation phase. Our modification entails identifying the inserted tokens in the syntax tree. The main effect of this strategy is that some of the terminal nodes of the syntax tree do not correspond to those present in the input stream. VASTapi represents them as children nodes of an error node.

Figure 4 shows an example of this strategy deleting tokens. As we can see in the input stream, there is an error in the function’s declaration header. The name of the function is repeated, the parser expects ‘(‘ but it finds the identifier test. The way of recovering from this error is removing
the erroneous token. Once the error is detected and visualized, the expected symbol is removed from the stack to allow the parsing process to continue.

### 3.3 Visualization of Panic Mode Strategy

Probably, the panic mode strategy is the most complicated one. It simultaneously and explicitly involves the stack, the input stream and the syntax tree. This strategy is based on synchronization points specified in the grammar. Each synchronization point is mapped to a group of synchronization tokens. Typical examples of these synchronization points are the end of language structures like body methods, sentences or parameter declarations. For example, in Java or C-like programming languages, their corresponding synchronization tokens would be ‘}’, ‘;’ and ‘)’.

The basic procedure of this strategy is as follows. When an error is detected, the parser searches for the next and closest synchronization point in the stack. If the parser does not end a synchronization point, the error recovery fails. Otherwise, the intermediate states and symbols between the error detection and the synchronization points are removed from the stack. Finally, the parser discards the tokens from the input stream until it receives a synchronization token.

Usually, this strategy is implemented in the generation tools using a special symbol. It is usually called the “error” symbol. This recovery method allows the easy definition of the synchronization points by inserting error symbols in the grammar, e.g., the rule:

```
METHOD DEC ::= TYPE ident ( error ) { BODY }
```

implements this error recovery strategy for a typical list of parameters. Thus, the synchronization tokens are those immediately after the error symbol, the “)” in the previous example.

In the specific case of the CUP parser generator, the panic error strategy is implemented using the
special symbol “error.” LALR parsers store in the stack symbols and states. Each state has information about what have been processed and what have to be processed next. When the parser detects an error in the input stream, it commutes to an internal error stage. Then, it searches within the stack the next state where the special symbol “error” has to be immediately processed, i.e., the next synchronization point. On the one hand, the recovery process fails if no synchronization point is found, i.e., the parser reaches the bottom of the stack during the search process. On the other hand, if a state with the “error” symbol has been located, the parser pops all the intermediate symbols and states between the top of the stack and the synchronization point, pushes the “error” symbol in the stack and performs the transition to the next state. All possible transitions from the current state will be performed with valid tokens, which are the corresponding synchronization tokens. Therefore, the parser discards from the input stream tokens until it receives a valid one, i.e., a synchronization token. If no synchronization token is received, the recovery process fails and the parser stops. Otherwise, the parser leaves the internal error stage and continues the parsing process normally.

3.3.1 Implementation of the Visualization

To master this strategy, one must be able to predict the contents of the stack, the remaining tokens in the input stream and the structure of the syntax tree just after the recovery process. Therefore, to enhance students’ comprehension of this strategy three different aspects should be visualized: the error detection, the discarded tokens from the input stream and the effect of popping contents from the stack. Actually, the contents popped from the stack are symbols and productions correctly processed but not connected with the syntax tree, because they are removed from the stack. Therefore they can be represented by partial sub-trees.

To visualize correctly these aspects, it was necessary to implement new functionalities in VASTapi. We have faced three challenges to visualize this error recovery strategy. First, when the parser enters in the internal error stage it does not perform any reduction; therefore VASTapi does not receive any information about the beginning of the recovery process. Second, the parser does not communicate the popped states and symbols from the stack when it searches the synchronization point. And third, while the parser is in the error stage, it does not identify the discarded symbols from the input stream.

The first challenge is to detect the beginning of the error recovery process. When the parser enters in the error stage, it calls to the syntax error method. CUP allows overriding this method in the parser specification. Therefore, this modification can be performed automatically and simultaneously with the parser specification annotation process (see Section 2). This allows VASTapi to detect the beginning of the error recovery process. Now VASTapi is able to mark tokens indicating whether they have been normally processed by the parser or they have been received during the error recovery process.

The rest of modifications performed in VASTapi are guided to identify the states and symbols popped from the stack and the tokens discarded from the input stream. The solutions to these two challenges are based on the detection of the end of the error recovery process and the location of the synchronization symbol. Both can be achieved with the reduction of the production where the synchronization point is located. Suppose that the following error production has been applied, \[ A ::= x \ y \text{error} \ z. \] This means that an error recovery process has been successfully performed. The symbols that have been popped from the stack are those correctly processed, reductions included, between the previous symbol to the ‘error’ one – ‘\(y\)’ in the previous example – and the
beginning of the error recovery process. The tokens discarded from the input stream are those received between the beginning of the error recovery process (previously marked by VASTapi) and the detection of the synchronization symbol – ‘z’ in the previous example.

Finally, the visualization of this process shows the three interesting elements of this strategy. First, the error production connected to the syntax tree. Second, a number of children nodes of the “error” node representing the partial sub trees, i.e., the tokens correctly processed but popped from the stack during the error recovery process. And three, a special node representing the tokens discarded from the input stream. Figure 5 shows a schematic visualization of this strategy.

3.3.2 Examples of the Visualization

We describe two different examples of the panic mode strategy and their visualization with VAST. We will use the same syntactical specification for all the examples, see Figure 6.

The first example shows a simple case where the parser recovers from one error. The input stream and the visualization of the recovery process are shown in Figure 7. The error is detected at line 1

```
S ::= E;
E ::= if ( COND ) { SENT } |
   if error ) { SENT };
COND ::= T OP T | T OP T and COND |
   T OP T error COND;
SENT ::= SENT EXP | EXP;
EXP ::= EXP2 | eval ( EXP2 ) | ident = EXP |
   eval ( error ) | error ( EXP2 );
EXP2 ::= EXP2 and L | L | EXP2 and error L;
L ::= T;
T ::= ident | cntnt;
OP ::= == | > | <;
```
after processing the `ident` token `n`, here the parser expects a conditional operator but it receives a `ctnt` token. The syntax rule used to recover from the error is `E ::= if error ) { SENT }`. The synchronization token is `close_p ')'`. The partial subtrees are made of the processed tokens `open_p '('` and `ident 'n'`. The tokens ignored from the input stream are displayed as children of the `IgnoredTokens` node. In this case, the discarded token is the unexpected `ctnt` symbol.

The second example shows two consecutive error recovery processes. The input stream and the corresponding visualization are shown in Figure 8. The first syntax error is detected at line 1 after processing the `ident` token `n`, here the parser expects a conditional operator as in the first example but it receives a `tand` token. The syntax rule used to recover from the error is `E ::= if error ) { SENT }`. The synchronization token is `close_p ')'`. The partial subtrees are made of the processed tokens `open_p` and a subtree whose root is the non terminal `T`. The tokens ignored from the input stream are displayed as children of the `IgnoredTokens` node. In this case, the discarded token are the unexpected symbols: `tand`, `ident 'q'` and `ctnt '1'`. The second syntax error is detected at line 2 after processing the `ctnt` token `20`, here the parser expects a `')'` token but it receives a `ctnt` token. The syntax rule used to recover from the error is `EXP ::= teval (`
Improving Students’ Performance with Visualization of Error Recovery Strategies in Syntax Analysis

The synchronization token is close_p ‘)’. The partial subtrees are made of the processed token ctnt. The tokens ignored from the input stream are displayed as children of the IgnoredTokens node. In this case, the discarded token is the unexpected ctnt symbol.

4. Evaluation

The typical features of VAST – syntax tree visualization and animation of its construction process – had been previously evaluated with positive results regarding the usability and its educational use, e.g., Almeida-Martínez and Urquiza-Fuente (2009).

We performed two different evaluations. Due to the simplicity of the error production and phrase level error recovery strategies, we conducted an observational study, mostly focused on usability issues. The panic mode strategy is significantly more complex than the previous strategies. Therefore, we conducted a more formal educational evaluation about it.

4.1 First Evaluation, an Observational Study

We conducted the first evaluation with experienced students who have passed the compilers course in previous years. The number of subjects was six and their participation was voluntary. This evaluation was focused on students’ opinion about ease of use, learning support to the syntax error recovery, quality of the tool and students’ satisfaction. Data were collected with a questionnaire and interviews.

During the experimental session, students completed two tasks with parsers developed with ANTLR and VAST. In the first task, students were provided with animations of specific cases of error recovery. We asked them to explain the details of the recovery processes visualized. In the second task, students were provided with a parser specification with error symbols inserted. We asked them to specify different input streams so that specific behaviours could be visualized, e.g., minimize the amount of input stream ignored while recovering from a specific error.

The answers to the questionnaire and the interviews allowed us to identify strengths and improvement points of the error recovery visualization with VAST. Ease of use and quality of the tool were the best graded aspects. The global view and the input stream highlighting were the two highest valued features by students. Two general improvements were detected: implementing tool tips for the playing tool bar and providing textual explanations to the visualizations. Finally, students commented that VAST should notify the user if the parser could not recover from syntax errors.

4.2 Second Evaluation, an Educational Study

This evaluation was focused on the panic mode strategy using the CUP tool. Subjects were 19 students enrolled in a Language Processors course from a CS degree. Their participation was voluntary and incentive based, i.e., they could slightly improve their course grades by 2% if they passed the course exam. To test the educational effect of the visualization of this strategy, we compared a typical scenario (control group, n=9) working just with the CUP parser generator tool, against the use of visualizations provided by VAST (treatment group, n=10). The groups were formed in a random/controlled way, i.e., we performed random selections of students ensuring that both groups were balanced in terms of a knowledge pre-test. Two dependent variables were used, students’ grades for the exercises and amount of exercises completed during the experimental session.

Before the experiment, students attended theoretical sessions regarding LL and LR parsing techniques. The error recovery strategies were explained within those sessions. Note that the treatment
and control groups were not differentiated during these theoretical sessions. Participants completed three tasks during the experimental session (two hours long). The first task consisted in reading some documentation about syntax error recovery extracted from well-known bibliography (Aho et al., 2007; Appel, 1998; Brown et al., 1992). The next task asked students to solve three problems with the same grammar specification. Students had to insert error symbols to develop the best recovery process for three different syntax errors. The last task consisted of modifying another grammar specification so the best recovery process is developed using fewer error symbols. The dependent variables were measured with the two later tasks.

Regarding students’ grades, the treatment group (M = 8.87; SD = 2.14) got significantly better grades, t(19) = -8.266, p < .01, than the control group (M = 1.74; SD = 1.52). This difference is too large, and can be explained by the second dependent variable, the amount of exercises completed. We detected that the treatment group completed many more exercises than the control group. After 40 minutes, all students of the treatment group had finished the first exercise, while only one student of the control group did it. At the end of the evaluation all the students of the treatment group had finished all the exercises while none of the control group did it. Therefore, VAST has allowed students to complete the exercises correctly and faster than others who just used CUP.

5. Related Work

There exist numerous tools to display some aspects of the compilation process. These tools can be classified into two groups. On the one hand there are tools focused on theoretical foundations. They animate some aspects of the compilation process such as the parsing table’s computing, input stream processing or syntax tree generation. However, these tools do not allow the user to generate his/her own parsers. The most representative example is JFLAP (Rodger, 2006). On the other hand, there are tools focused on practice. They provide the students with visualizations of some aspects of their own parsers. In addition, some of these tools allow to generate parsers (Bovet, 2011; Kaplan and Shoup, 2000; Krebs and Schmitz, 2004; Lovato and Kleyn, 1995; Mernik and Zumer, 2003; Resler and Deaver, 1998; Sierra et al., 2008), but others just produce visualizations (Andrews et al., 1988; Gómez-Martín et al., 2006; Khuri and Sugono, 1998; Resler, 1990; Vegdahl, 2001). All of these tools have two remarkable problems: they only work with a particular generation tool and they only display certain parts of the compilation process.

There is no tool that covers the entire parsing algorithm and visualizes all the parsing related aspects. Therefore, a teacher has to use more than one tool, switching between different notations, organizations and visualizations. Thus, students have to learn how to use quite different tools: specification notation, construction process, interpretation of output messages – conflict reports, transitions matrix or items sets. Furthermore, the teacher has to dedicate time to become familiar with the different tools, and to plan their integration in the course. This makes more difficult their use in educational environments (Naps et al., 2003).

VAST fits with the tools focused on practice that allow students to generate their own parser. Due to its generic approach, it can be used with different generating tools and different parsing techniques. In addition, its design allows to augment its visualization features to cover different aspects related to the parsing techniques visualized. Finally, the authors have not found any formal evaluation of the previously described tools. VAST has been evaluated from the usability and educational point of views (Almeida and Urquiza, 2009), providing empirical support to its visualization features.

Regarding the visualization of syntax error recovery strategies, we have not found any tools that
visualize all the syntax error recovery strategies. Only the tool ANTLRWorks (Bovet, 2011) does something similar, but their visualizations are dedicated to error detection – showing partial syntax trees – rather than error recovery visualizations. Furthermore, we have not found any evaluation of these visualizations. VAST produces visualizations for all the syntax error recovery strategies. In the specific case of the panic mode strategy, it has been demonstrated that VAST improves students’ performance developing error recovery specifications. Students who used VAST were able to develop correct error recovery specifications faster than students who just used the parser generator tool.

6. Conclusions
Parser visualization is not a novel approach to improve students’ comprehension of syntax analysis. But few efforts have been dedicated to such a complex aspect as syntax error recovery. In this work we present the design and implementation of an educational tool that visualizes this aspect. To this end, we have used VAST, an existing tool focused on syntax analysis visualization. We have successfully evaluated the effectiveness of these visualizations in the context of a Language Processors course. We have compared the use of VAST against a typical scenario where only common parser generation tools are used. We can conclude that the use of VAST has improved students’ developing performance of syntax specifications with error recovery.

Acknowledgement
This work is supported by project TIN2011-29542-C02-01 of the Spanish Ministry of Science and Innovation.

References
Improving Students’ Performance with Visualization of Error Recovery Strategies in Syntax Analysis


Biographical Notes

Jaime Urquiza-Fuentes (http://www.escet.urjc.es/~jurquiza/) is an associate professor at the Technical Superior School of Computer Science Engineering of the Universidad Rey Juan Carlos. He received his BSc in computer science from the Universidad Politécnica de Madrid (1999) and PhD degree in computer science from the Universidad Rey Juan Carlos (2007). His research interests are educational technologies, software visualization, information visualization and human-computer interaction.

Francisco J. Almeida Martínez, was born in Las Palmas de Gran Canaria in 1984. In 2006 he finished computer science systems studies in Las Palmas de Gran Canaria University. In 2008 he got his BSc in computer science from Rey Juan Carlos University. Afterwards, in 2011 he finished his PhD in computer science at Rey Juan Carlos University. He has worked in the Laboratory of Information Technologies in Education for more than five years. His main areas of research are educational software, information visualization and educational visualization.

Antonio Pérez-Carrasco is currently working as a teacher at the International University of La Rioja and as a software engineer for ACS Group. He worked as a teacher and a researcher for six years at Rey Juan Carlos University. He became a Doctor of Computer Science in December 2011. His research interests include programs visualization, software aimed at computer science education and web technologies.
J. Ángel Velázquez-Iturbide received his computer science degree and PhD degree in computer science from the Universidad Politècnica de Madrid, Spain, in 1985 and 1990, respectively. He is currently with the Universidad Rey Juan Carlos as a professor, where he is the Director of the Department of Computing Languages and Systems I. His research areas are educational software and instruction for programming education, software visualization, and human-computer interaction. Professor Velázquez is an affiliate member of IEEE Computer Society and IEEE Education Society, and a member of ACM and ACM SIGCSE. He is the President of the Spanish Association for the Advancement of Computers in Education (ADIE).