Journal of Universal Computer Science, vol. 15, no. 7 (2009), 1546-1561
submitted: 28/11/08, accepted: 28/3/09, appeared: 1/4/09 © J.UCS

Visualization of Syntax Trees for Language Processing Courses

Francisco J. Almeida-Martínez, Jaime Urquiza-Fuentes
J. Ángel Velázquez-Iturbide
(Universidad Rey Juan Carlos, Madrid, Spain
{francisco.almeida,jaime.urquiza,angel.velazquez}@urjc.es)

Abstract: This article describes the educational tool VAST. We designed VAST to be used in compiler and language processing courses. The current version allows generating and visualizing syntax trees and their construction process. The main advantages of VAST follow: it is designed to be as independent from the parser generator as possible, it allows students to visualize the behavior of parsers they develop, and it has an interface designed to easily handle huge syntax trees. Finally, we describe two satisfactory preliminary evaluations from the usability and educational points of view.

Key Words: Syntax Trees, Visualization, Educational Software
Category: D.3.4, K.3.1, K.3.2

1 Introduction

Language Processors (LP) and *Compilers* are often perceived by students as some of the most complex subjects. Typical topics covered by these subjects are: the scanning and parsing phases, syntax directed translation and, if the subject is compilers, symbol tables, semantic analysis, and intermediate and object code generation. The scanning and parsing phases are clearly based on formal languages theory. Syntax directed translation and its use in compilers –semantic analysis and intermediate code generation- do not have such a clear binding with formal languages theory, but they require a clear understanding of the underlying syntax structure.

Automatic parser generators have assisted in reducing the complexity of LP design. These tools produce LP code from a user specification –lexical, syntactic or translation–, see the well-known pair *Lex* & *Yacc* [Levine et al. 1992]. These tools are used in educational contexts, but they are professional tools rather than educational tools. Acquiring expertise in these tools is an additional advantage, but they are not easy to learn. Thus, the complexity of LP courses is increased by the use of these tools. Furthermore, there exist many parser generators, but they have not a homogeneous way to specify a LP, the most notable differences being the notation and organization of syntactic and lexical specifications.

The scanning phase exhibits a close relationship between the theoretical foundations –finite state automaton and regular expressions– and its corresponding generation tools –that associate actions to patterns, quite similar to regular expressions–. Consequently, its learning curve is smooth. The case of the parsing phase is different. The relationship between the theoretical foundations –stack automaton and context free grammars– and its corresponding generation tools –that associate actions to grammar rules– seems to be close again. However, there are other topics in the subject, closely related to syntax but without specific support from generator tools. Some examples are action and go-to tables of LR parsers, the error recovery process or syntax trees (ST). The two former topics require the assistance of an expert on the tool; the latter one is not supported by these tools. Notice that the understanding of STs generated and their building process is very important for the most complex part of the subject, namely syntax directed translation.

In this article we present VAST[1], an educational system designed to visualize STs. With VAST, students are capable to watch the ST generated by their own parsers, filling a gap between theory and practice of LP design. This is not a novel approach, but existing solutions are partial or too specific. VAST solves this problem from a more sound and generic approach.

The rest of the article is structured as follows. In the Section 2 we describe related work. In Section 3 we explain the implementation of VAST. Then, we show the different visualizations generated by VAST in Section 4. In Section 5 we describe its educational use and the evaluations that we have performed at the moment. Finally, we state our conclusions and future work in the Section 6.

2 Related Work

As we have previously mentioned, there exist tools focused on filling the gap between theory and practice of syntax analysis, but they do it in a partial or too particular way. We survey these tools in this section.

On one side of the gap we have found the educational tool JFlap [Rodger 2006]. This tool represents a valid approach for the theoretical foundations, even its design has a clear educational aim. With JFlap, students are asked to simulate the construction process of an LR automaton, the generation of the *head* and *follow* sets or the processing of a given input stream –visualizing the ST– among other possibilities. However, this system does not allow the students to generate their own parsers.

On the other side of the gap, we have found tools that visualize the matching process from a more practical point of view. We have found nine tools, summarized in table 1. For each tool we detail the parsing algorithm/s supported, the visualization of the algorithmic behavior, the visualization of the ST generated and the possibility of generate parsers specified by the user.

[1] Available at http://www.lite.etsii.urjc.es/vast/

Table 1: Summary of visualization tools related with parsers

Tool	Parsing algorithm	Parser	Tree	Generator
ICOMP [Andrews et al. 1988]	LL(1)	$\star$	$\star$	
VisiCLANG [Resler 1990]	LL(1)	Grammar		
APA [Khuri and Sunogo 1998]	SLR(1)/LL(1)	$\star$	$\star$	
TREE-VIEWER [Vegdahl 2001]	not specified		$\star$	
VCOCO [Resler and Deaver 1998]	LL(1)	Grammar		COCO/R
GYacc [Lovato and Kleyn 1995]	LALR(1)	$\star$	$\star$	Yacc
CUPV [Kaplan and Shoup 2002]	LALR(1)	$\star$		Cup
LISA [Mernik ans Zumer 2003]	SLR-LALR(1)/LL(1)		$\star$	own
ANTLRWorks [Bovet 2008]	LL(k)		$\star$	ANTLR

The *parsing algorithms* visualized are: LL(1), LL(k), SLR(1) and LALR(1). The column labeled *Parser* specifies how the algorithmic behavior is visualized, there are two ways: showing transitions in the tables with the stack and the input stream ($\star$), or just highlighting the grammar rule applied (Grammar). The column labeled *Tree* specifies if the tool visualizes the ST. And finally, the column labeled *Generator* identifies the parser generator used by the tool, if any.

The first four tools do not allow to generate parsers specified by the user. Thus, ICOMP [Andrews et al. 1988] and VisiCLANG [Resler 1990] are developed for a concrete language, while APA[2] [Khuri and Sunogo 1998] and TREE-VIEWER [Vegdahl 2001] are just visualization tools.

VCOCO [Resler and Deaver 1998] allows to generate user specified parsers, it uses the tool COCO/R [Mössenböck 1990]. Its visualization highlights the current execution point in: the grammar, the lexical and syntactic specifications, the input stream and the source code of the generated parser. This tool represents a debug approach for experts rather than an educational tool for students, this is why we put it in a separate row.

The last four tools give more elaborated visualizations and allow to generate user specified parsers. Only one of them, CUPV [Kaplan and Shoup 2002], does not visualize the ST. However, all of them are highly dependent from an own notation or the generation system used. This approach restricts its educational use because the parser and the visualization system are totally dependent.

There is not any tool that covers all the parsing algorithms and visualizes all the dimensions –algorithmic behaviour and ST–, therefore it is possible that a teacher has to use more than one, switching between different notations, organizations and visualizations. In this context, the students have to learn how to use 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

[2] Actually, this tool has not name, we have used the acronym of the title of the article where it is described.

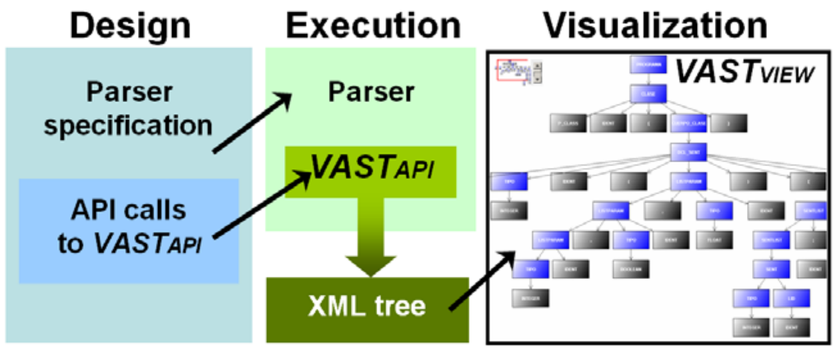


Figure 1: The functional scheme of VAST

environments, and to plan their integration in the course. This could make more difficult their use in educational environments [Naps et al. 2002].

Finally, we have not found any tool covering the syntactic error recovery. We think that this topic is complex enough to require some visual support.

Therefore, we will focus our efforts on filling the gap between theory and practice. We will visualize the ST and its building process with a visual interface specially designed for this task. We will keep as independent as possible the parser specification and the ST visualization.

3 The Implementation of VAST

Our main concern for the implementation of VAST was to separate the ST visualization and its building process. To this aim, VAST offers an API –VASTApi– designed to be used when the parser is building the ST, and a graphical interface –VASTview– to visualize the ST generated. VASTApi translates grammar rule application events to information that will be displayed by VASTview. This structure ensures a high degree of independence between VAST and the parser generator used. See Figure 1 for a schematic view of the design of VAST and its use.

The information produced by VASTApi is XML-based, thus we have used *JDOM* [Hunter 2008] for the implementation of VAST. VASTApi allows generating XML information from each grammar rule application; therefore, the XML information will not be complete until the parser finishes its execution. At the moment, we have implemented support for bottom-up parsers.

We have implemented VASTview using the visualization software *JGraph* and *JGraphPro* [JGraph 2008]. We have used JGraph for building the nodes of the tree and JGraphPro for drawing the tree.

Listing 1: An example of a CUP parser specification annotated with API calls

```
1  public xmlIntermiddleLR Vv = xmlIntermiddleLR();
2
3  PROGRAM ::= CLASS
4  {: parser.Vv.setRoot("PROGRAM");
5  parser.Vv.addProduction("PROGRAM","CLASS");
6  :};
7
8  CLASS ::= RWCLASS IDENT OPEN_BR CLASSBODY CLOSE_BR {:
9  parser.Vv.addProduction("CLASS","RWCLASS_IDENT{CLASSBODY}");
10  :};
```

4 Working with VAST

In this section, we explain the basic use of VAST. First, the user has to annotate her/his parser specification using methods from VASTApi. Once the parser has been generated, its execution will produce the information used by VASTview to visualize the ST and animate its construction process.

4.1 Generating the Visualizations with VASTapi

The main aim of VAST is to allow the generation and manipulation of the ST independently from the parser generator. The parser specification has to be adapted inserting API calls in it, see an example in listing 1. Due to the API calls, the execution of the parser generates an intermediate XML based representation of the ST. This XML information is the data source of VASTview.

VASTApi require some information to work properly, this information must be provided by the user. API calls are inserted as part of the typical actions associated to grammar rules. The information needed by the API is just the grammar rule applied, using the method `addProduction` (lines 6 and 11 in the listing 1).

Taking into account bottom-up parsers, once the axiom has been reduced, VASTApi must be informed that the root has been fixed and the process of the input stream has finished. This is done with the `setRoot` method (line 5 in listing 1)[3]. Now VASTApi can build the XML intermediate representation of the ST that will be visualized by VASTview, see an example in appendix A.

[3] Note that API calls are performed through an object that must be previously declared (see line 1 in the listing 1)

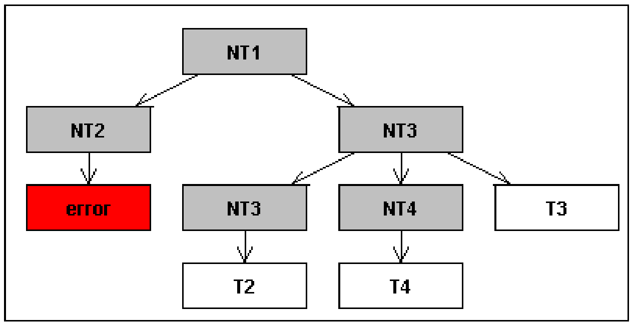


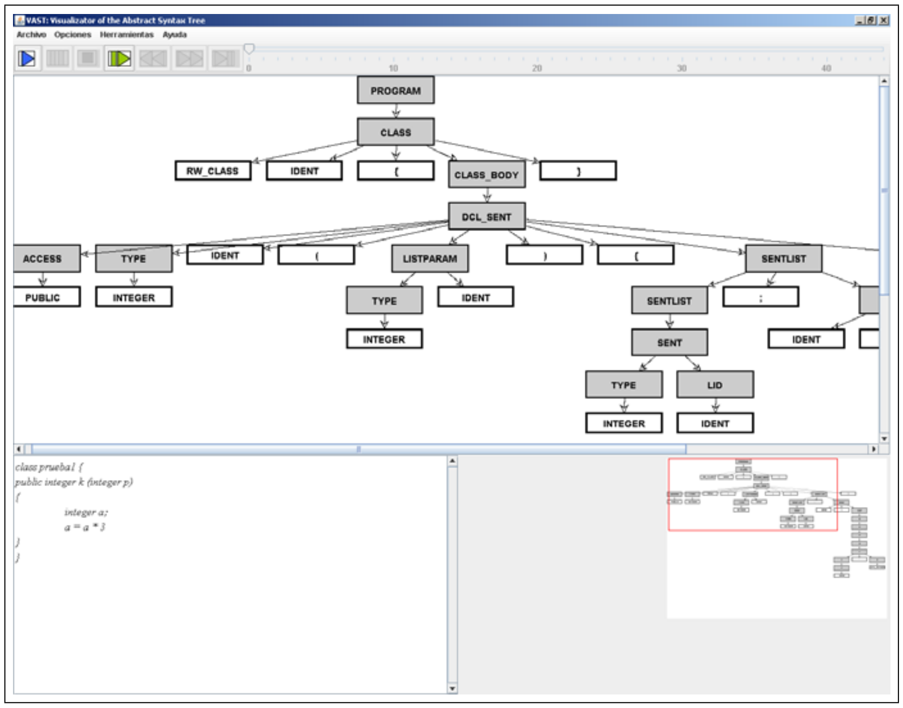
Figure 2: Graphical representation of terminals (Tx), non-terminals (NTx) and error recovery places (error)

4.2 Visualizing the ST with VASTview: Graphical Representation and Manipulation Interface

The graphical representation of the ST is the hierarchical structure resulting from the grammar rules application, which is a tree. We allow to give different representations to terminal nodes (T), non-terminal nodes (NT) and error nodes, see Figure 2. The T nodes are the leaves of the tree, the NT nodes are internal ones, and error nodes represent a place where the parser has recovered from a syntactic error. The error nodes only appear if the user has included error recovery inside the parser specification. With the visualization of the error nodes, the user can see the exact error recovery place and the amount of the input stream correctly processed.

We have designed VAST thinking about parsers designed by students. Probably, the ST produced by these parsers are big and without any fixed structure (symmetry, with or height). Therefore, we have developed VASTview, a graphical interface specially designed to cope with such trees. This interface is made up of a global view and a detailed view of the ST, together with zoom actions, subtree aggregation and animation of the construction process of the ST.

The global and detailed views allow the user to easily manipulate the ST, see Figure 3. The global view shows the whole ST highlighting its visible part in detailed view. The detailed view facilitates the students a closer inspection of the ST with zooming, aggregating and scrolling, all of them synchronized with the global view. Zooming actions on the detailed view allow the students to focus their attention on specific parts of the ST adjusting the desired level of detail. Subtree aggregation –by means of expand/collapse actions– allow the students to maintain a representation of the ST where only interesting parts of it are visible, see Figure 4. Finally, scrolling allows students to watch every node in



```
class prueba1 {
public integer k (integer p)
{
        integer a;
        a = a * 3
}
}
```

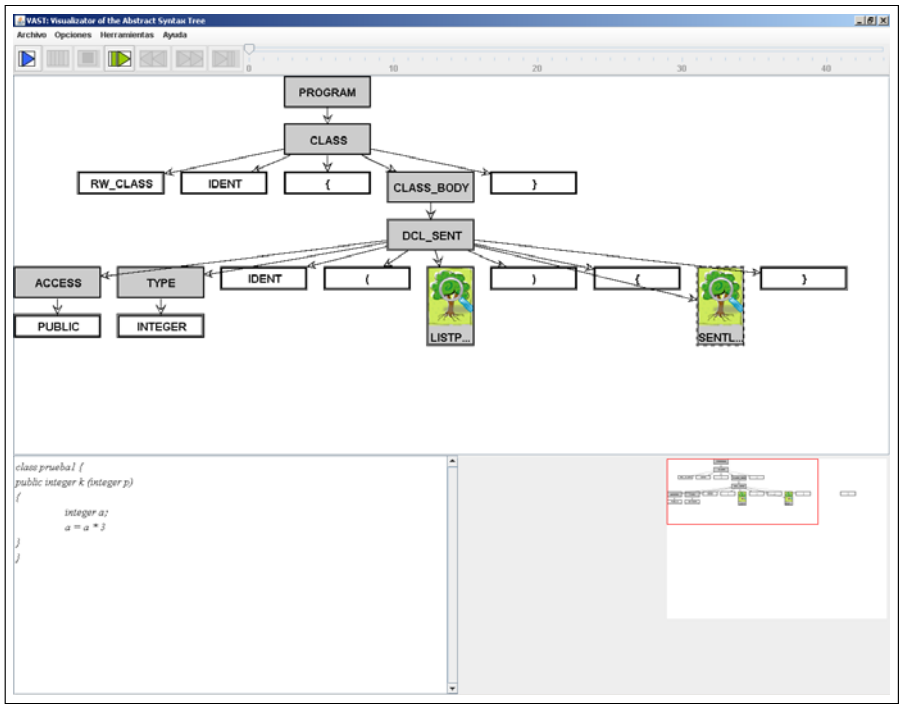
Figure 3: ST manipulation interface of VASTview. The highlighted area in the global view is the visible part of the ST available in the detailed view

the tree changing the portion of the tree visible in the detailed view. Scrolling can be performed with the scroll bars of the detailed view and directly moving the highlighted area in the global view.

4.3 Animating the Construction Process of the ST with VASTview

We animate the construction process of the ST using the different intermediate stages. As we have said previously, we only work with bottom-up parsers at the moment. The animation of the construction process help students to see how the input stream is matched by means of shifts –terminal node creation– and reductions –connection of existing nodes with a new non-terminal node–, together with the error recovery. Playing an animation is as easy as using typical VCR controls, together with a slide bar allowing fast location of specific stages of the matching process.

The ST changes its shape, area and contents during its construction process. Thus the interface could adapt to each stage using a best-fit policy changing the location of the nodes and other properties of the graphical representation. We

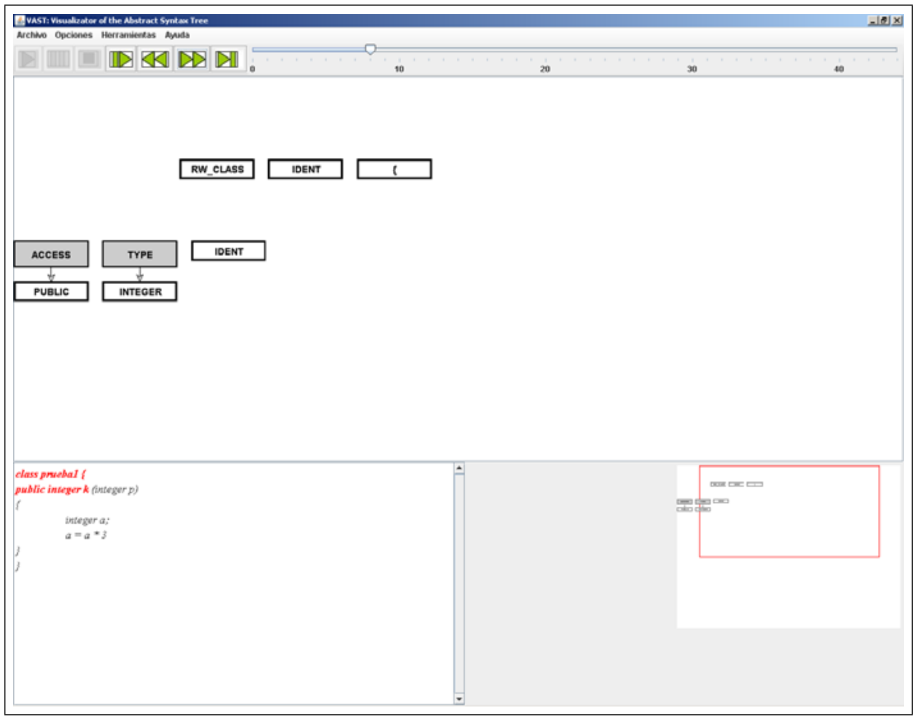


```
class prueba1 {
public integer k (integer p)
{
        integer a;
        a = a * 3
}
}
```

Figure 4: An example of subtree aggregation with the ST in the Figure 3. Here the non-terminals `LISTPARAM` and `SENTLIST` have been collapsed

have decided to maintain these properties unless the student changes them. All the nodes keep their location from their creation to the end of the process. This prevents students from distracting, e.g. while searching for the new location of existing nodes.

Furthermore, to make the animation more meaningful we also visualize the input stream. We highlight the last processed terminal, and therefore the processed and unprocessed parts of the input stream. The figures 5 to 7 show how the construction process of the ST is animated. Figure 5 shows an early stage of the matching process where six terminals have been processed and only two reductions have been applied. Figure 6 shows an intermediate stage where all but two terminals have been processed, but there are still many reductions (12 out of 23) that have to be applied. Finally, Figure 7 shows the whole ST, with all the input processed and all the reductions applied.


VAST: Visualizator of the Abstract Syntax Tree
Archivo Opciones Herramientas Ayuda
0 10 20 30 40
RW_CLASS
IDENT
{
ACCESS
TYPE
IDENT
PUBLIC
INTEGER

```
class prueba1 {
public integer k (integer p)
{
            integer a;
            a = a * 3
}
}
```

Figure 5: An early stage of the matching process with six terminals shifted and two reductions applied

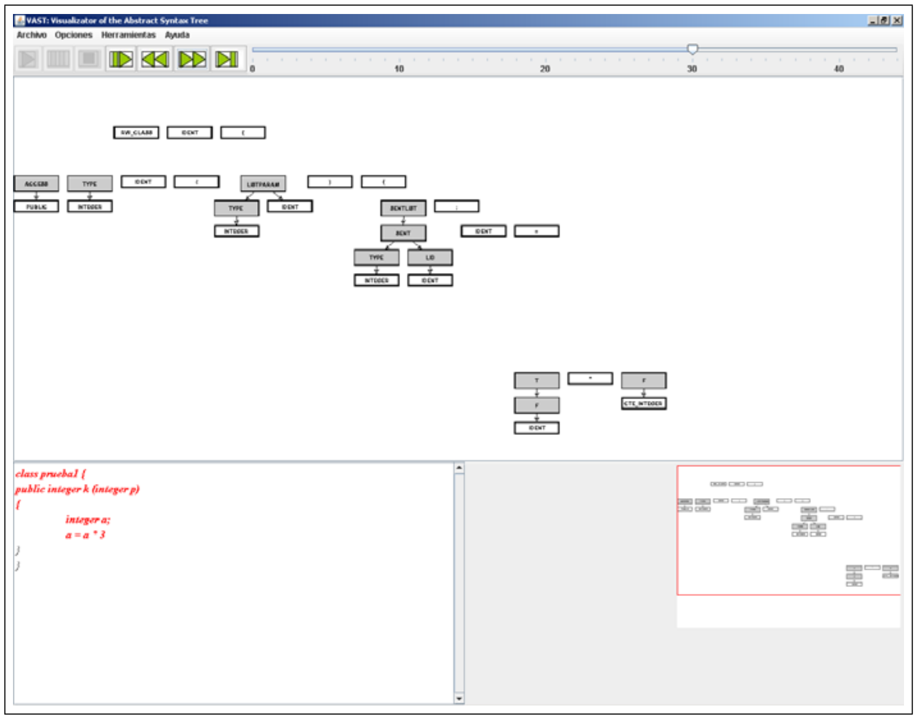
5 Educational use of VAST

VAST allows students to view and manipulate an ST and its construction process. To this end, students have to insert simple API calls within their parser specification. In a related field, literature about algorithm visualization with educational aims [Hundhausen et al. 2002] has shown that active use of visualizations by students improves their learning process. Thus, visualizations become a part of the student educational experience rather than the main element of this experience.

In this section we illustrate the active use of the visualizations generated with VAST. Due to the low number of students available for the evaluation, only eleven, we have used a single group pre-post test design.

5.1 Subjects and Protocol

The participation was voluntary, all the subjects were male. They were enrolled in a language processors course of the fourth year of a (five years) BSc in Com-



```
class prueba1 {
public integer k (integer p)
{
            integer a;
            a = a * 3
}
}
```

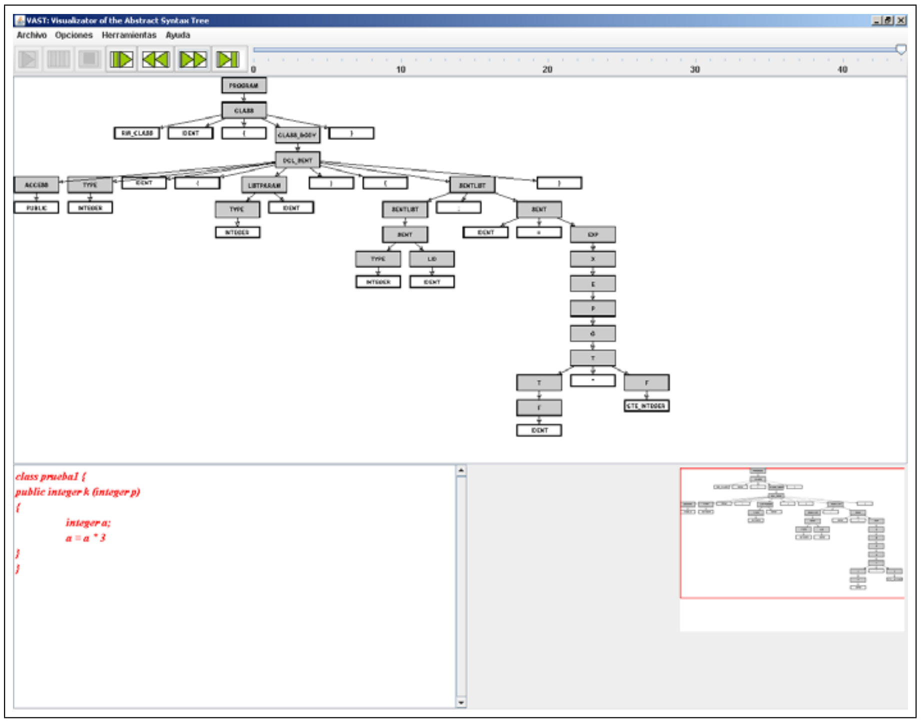
Figure 6: An intermediate stage with most of the input shifted but more than the half of the reductions pending to be applied

puter Science at the Universidad Rey Juan Carlos[4].

The context of the evaluation was LL(1) parsers. Students completed the pretest after the last theoretical session, two weeks before the evaluation. Note that VAST needs a parser generation tool. Therefore we ensured previous knowledge of the parser generation phase using ANTLR[5]. One week before the experiment, students had a lab session with exercises about the ANTLR parser generator so they were familiar with ANTLR syntax for grammar specifications and their generation process.

The evaluation was two hours long. First, the instructor gave a tutorial about the VAST and its use with ANTLR. Then, the students worked with the exercises. Students completed the post test, see appendix B, in the next session after the evaluation. The questions of the post test were mapped to four of the Bloom's taxonomy levels [Bloom et al. 1959]: knowledge (questions 1, 2 and 3), comprehension (question 4), application (question 5) and synthesis (question 6).

[4] http://www.urjc.es
[5] http://www.antlr.org/, 2009



```
class prueba1 {
public integer k (integer p)
{
        integer a;
        a = a * 3
}
}
```

Figure 7: The last stage having completed the matching process

5.2 Exercises

Students have to code the syntax specification using a general purpose editor. Then they annotate the specification using simple VASTApi calls. Next they generate the parser using ANTLR with the annotated specification and compile the generated parser. Each execution of the parser –using the console– produces the visualization that can be viewed with VASTView.

In the first exercise students were asked to change a grammar of arithmetic expressions so existing precedences of binary operators were changed. The original grammar was:

```
S := F N
N ::= + F N | - F N | * F N | / F N | λ
F ::= id | cte | ( S )
```

The new grammar must give the highest precedence to `*` and `/` operators –both the same–, then a medium precedence to the `-` operator and the lowest precedence to the `+` operator. And all binary operators must be right-associative.

The solution to this exercise is a grammar together with: a screen shot (or a group of them) showing the existing precedences, the input stream(s) used to

Table 2: Post test results

Bloom's level	Grades [0-1]	Tasks
Knowledge	0,62	Define concepts and enumerate features
Comprehension	0,89	Perform single steps of the parsing process
Application	0,91	Simulate part of the parsing process
Synthesis	0,68	Create an LL(1) grammar with specific features

produce the visualization(s) and a textual explanation of the solution.

In the second exercise, students were asked to generate two inputs with syntax errors produced by problems with: the *starters* symbols and the expected terminals inside right parts of the grammar rules. Again, the solution to both cases is made up of the input stream, the visualization and the textual explanation.

Finally, in the third exercise students were asked about syntax error recovery situations with the panic mode strategy. For each non terminal symbol `X` in the grammar, students have to generate an erroneous input stream were the group of terminals in `FOLLOW(X)` were used as the synch terminals. Again, the solution is made up of the input streams, the visualizations and the textual explanations.

5.3 Results

Pretest results showed that all the students have similar knowledge about LL(1) parsers. Students' feedback about VAST was positive, they think that VAST is easy to use and helps them in the learning process. The post test results were also positive, the Table 2 shows them grouped by each taxonomy level, the grades –in a [0-1] scale– and a description of the tasks involved.

5.4 Other Usability Evaluations

At the moment, we have evaluated VAST with an expert review and an observational evaluation. After the expert review, we improved the tree manipulation interface of VAST and the multiple views available while playing the construction process.

The observational evaluation required the voluntary participation of students. Students were enrolled in a subject of *compilers and interpreters*, in the last year of a three years CS degree. The evaluation was divided in two sessions. The first session was focused on training, where the teacher introduced VAST. In the second session (one week after), students were asked to generate their own visualizations using CUP [Hudson et al. 2008] and VAST. Students thought that the experience was quite satisfactory. Their opinions showed that VAST is easy to use, but we also detected some mistakes, e.g. forgetting to fix the root node. Due to this kind of mistakes, we are thinking of automating the insertion of API calls within the parser specification.

6 Conclusions and Future work

We have presented the educational tool VAST, aimed at the visualization of syntax trees. We have identified a large gap existing between concepts taught in theory and generation tools used in practice. We feel that this gap can be filled by the visualization of STs and their construction process. Moreover, visualization of STs may assist in learning/teaching syntax directed translation.

We have surveyed relevant, related tools. Tools that allow users to generate their own parsers either demand high expertise or are tightly coupled to a given environment. Actually, every tool has its own way to specify the parser, report errors or show transition tables.

We have created VAST to solve these problems. Visualizing an ST and its construction process is almost independent from the parser generator adopted. Thus, a teacher can choose a parser generator based on her/his own criteria –parsing algorithm, specification format– and then use VAST to visualize STs. We want to highlight that VAST was developed so that two parts are isolated: the generation API –VASTapi– and the visual interface –VASTview. VASTapi was designed to build STs, its output being an XML file. VASTview interprets such an XML file to visualize the ST and its construction process. Therefore we have two independence levels: one between the parser generator and VASTapi, and the other between VASTapi and VASTview. At the moment we have developed VASTapi with Java, so we are not totally independent from the parser generator. However, just porting our API to other language will enable to use VAST in other development environments.

We have evaluated VAST twice. We have conducted usability inspections with an expert review and an observational evaluation. Students opinion was positive and we obtained useful information for further improvements. Also, we have performed a pedagogical evaluation, its results show that students reach acceptable grades regarding four different Bloom's taxonomy levels. We realize that the generalization of these results is limited due to the single group design and the low number of participants. But we, and also the students, feel that the tool help them in the learning process, and this evaluation give us hints on how to continue the development of VAST.

Future works can be classified in two categories: interactive and educational. Future work on interaction will be focused on improving how students use the tool. We plan to develop a configuration utility for visual elements. We will improve navigation through the parsing process, allowing students to select pieces of the input stream and showing the corresponding state in the parsing process. Finally, we would like to support the automatic insertion of API calls in the parser specification.

Educational future work is focused on improving the features of visualizations and animations as educational resources. Thus, students could see simultaneously the construction process, the grammar rules applied, and some textual explanations of the different actions performed.

Acknowledgements

This project is supported by projects TIN2008-04103/TSI of the Spanish Ministry of Education and Science.

References

[Andrews et al. 1988] Andrews, K.; Henry, R. R; Yamamoto, W. K.: “Design and Implementation of the UW Illustrated Compiler”; SIGPLAN Not. 23, 7 (June 1988), 105–114.

[Bloom et al. 1959] Bloom, B.; Furst, E.; Hill, W.; Krathwohl, D.R.: “Taxonomy of Educational Objetives: Handbook I, The Cognitive Domain”; Addison-Wesley.

[Bovet 2008] Bovet, J.: “ANTLRWorks: The ANTLR GUI development environment”. (2008) `http://www.antlr.org/works/index.html`.

[Hudson et al. 2008] Hudson, S.; Flannery, F.; Ananian, C. S.: “CUP LALR parser generator for Java”. (2008) `http://www2.cs.tum.edu/projects/cup/`.

[Hundhausen et al. 2002] Hundhausen, C.; Douglas, S. ; Stasko, J.: “A meta-study of algorithm visualization effectiveness”; J. of Vis. Lang. and Comp. 13, 3 (June 2002), 259–290.

[Hunter 2008] Hunter, J.: Jdom. (2008) `http://www.jdom.org/`.

[JGraph 2008] Java graph visualization and layout. (2008) `http://www.jgraph.com/`.

[Kaplan and Shoup 2002] Kaplan, A.; Shoup, D.: “CUPV a visualization tool for generated parsers”; SIGCSE Bull 32, 1 (March 2000), 11–15.

[Khuri and Sunogo 1998] Khuri, S.; Sugono, Y.: “Animating parsing algorithms”; SIGCSE Bull. 30, 1 (March 1998), 232–236.

[Levine et al. 1992] Levine, J. R.; Mason, T.; Brown, D.: “Lex & Yacc”; O’Reilly.

[Lovato and Kleyn 1995] Lovato, M. E.; Kleyn, M. F.: “Parser visualizations for developing grammars with Yacc”; SIGCSE Bull 27, 1 (March 1995), 345–349.

[Mernik ans Zumer 2003] Mernik, M.; Zumer, V.: “An educational tool for teaching compiler construction”; IEEE T. Educ. 46, 1 (Feb 2003), 61–68.

[Mössenböck 1990] Mössenböck, H.: “A generator for production quality compilers”; Proc. 3rd Intl. W. Compiler Compilers CC’90, Lec. Notes Comp. Sci. 477, Springer-Verlag, New York (1990).

[Naps et al. 2002] Naps, T.; Rößling, G.; Almstrum, V.; Dann, W.; Fleischer, R.; Hundhausen, C.; Korhonen, A.; Malmi, L.; McNally, M.; Rodger, S.; Velzquez-Iturbide, J.: “ITICSE 2002 working group report: Exploring the role of visualization and engagement in computer science education”; SIGCSE Bull. 35, 2 (June 2002), 131–152.

[Resler 1990] Resler, D.: “VisiCLANG—a visible compiler for CLANG”; SIGPLAN Not. 25, 8 (August 1990), 120–123.

[Resler and Deaver 1998] Resler, R. D.; Deaver, D. M.: “VCOCO: a visualisation tool for teaching compilers”; SIGCSE Bull. 30, 3 (September 1998), 199–202.

[Rodger 2006] Rodger, S.: “Learning automata and formal languages interactively with JFLAP”; SIGCSE Bull. 38, 3 (September 2006), 360–360.
[Vegdahl 2001] Vegdahl, S. R.: “Using visualization tools to teach compiler design”; J. Comput. Small Coll. 16, 2 (January 2001), 72–83.

A An example of the XML intermediate representation of the ST

```
<?xml version = "1.0" encoding = "UTF-8"?>
<!DOCTYPE tree SYSTEM "file:../vast/XML/tree.dtd">
<root text ="PROGRAM" position="44" fileToParse="test1.txt">
<node text ="CLASS" position="43" line="-1" column="0" lex="">
    <node text ="RW_CLASS" position="1" line="1" column="6"
    lex="class"> </node>
    <node text ="IDENT" position="2" line="1" column="14"
    lex="prueba1"> </node>
    <node text ="{" position="3" line="1" column="16" lex="{">
    </node>
    <node text ="CLASS_BODY" position="41" line="-1" column="0"
    lex="">
        <node text ="DCL_SENT" position="40" line="-1"
        column="0" lex="">
            <node text ="ACCESS" position="5" line="-1"
            column="0" lex="">
                <node text ="PUBLIC" position="4" line="2"
                column="7" lex="public"> </node>
            </node>
                    ....................
</node>
</root>
```

B Post test used in the evaluation

1. What are the main characteristics of LL(1) parsers?
2. What is a derivation?
3. How would you define the panic error recovery?
4. Given the following grammar:
   `S ::= F S’`
   `S’ ::= + F S’ | - F S’ |lambda`
   `F ::= id | cte | ( S )`
   Show the cases of a non-recursive LL(1) parser, indicating the states (input and stack) before and after the following operations: (a) derivation, (b) token recognition and (c) recover from an error using panic error recovery with the Follow (A), where $A \in \{A, S', F\}$ as synchronization tokens

5. Using the same grammar as in the previous exercise:
   ```
   S ::= F S'
   S' ::= + F S' | - F S' |lambda
   F ::= id | cte | ( S )
   ```

   It has been developed a non-recursive LL(1) parser to this grammar with a panic error recovery. The synchronization tokens are the following of the antecedent which is being processed. Suppose that the parser is in the state: $\mathbf{\$}(S')S, idctecte) - cte)\mathbf{\$}$. Indicate the following steps of the parser until the stack only contains the symbol S'.

6. Design a LL(1) grammar which can recognize logic expressions with the following characteristics:

   – Operands are the logic constants true or false.

   – Operators are the parenthesis, "not", "xor", "and", "or", "nand" and "or". The binary operators associativity is on the right. The precedente is specified as follow:

()	+ **The biggest precedence**
not	
and or	Both the same precedence
or nand	Both the same precedente
xor	**- The lowest precedente**