Educational visualizations of syntax error recovery

Francisco J. Almeida-Martínez, Jaime Urquiza-Fuentes, J. Ángel Velázquez-Iturbide
LITE - Laboratory of Information Technologies in Education. Depto. Lenguajes y Sistemas Informáticos I
Universidad Rey Juan Carlos
Madrid, Spain
{francisco.almeida,jaime.urquiza,angel.velazquez} @urjc.es

Abstract— **This work is focused on the syntax error recovery visualization within the compilation process. We have observed that none of the existing tools, which display some views of the compilation, give a solution to this aspect. We present an educational tool called VAST which allows to visualize the different views of the compilation process. Besides, VAST allows to display different syntax error recovery strategies.**

Educational technology; Visualization; Computer science education; Languages

I. Introduction

Language processors or compilers are considered by the students as ones of the most difficult subjects in Computer Science degree. One of the most difficult parts in these subjects is the syntax error recovery.

In the syntax analysis points out the syntax tree (ST) comprehension. The ST concept is basic for syntax analysis and syntax directed translation. The understanding of both topics can be improved using visual representations of the ST. Moreover, the tree representation of the ST is quite similar to student's mental representations of the ST.

Card et al. [7] defined visualization as the "use of computer-based, interactive visual representations of data to amplify cognition". Visualizations, which show the behavior of the internal parsers structures, can be used to make easier the comprehension of the syntax analysis. Nowadays, we have not found any tool which displays the syntax error recovery.

Some syntax error recovery (SER) strategies are difficult to understand for students. Also their implementations in the parser generating tools (e.g. The panic mode error recovery of Yacc): "Proper placement of error tokens in a grammar is a black art…" [13].

Lexical error recovery (LER) is simpler than SER. LER consist in transforming an unmatched string of characters in a valid string using single character modifications (insertion, deletion or modification) or just reporting the error and then restart the lexical analysis process with the next character. However, SER has more implications; sometimes it does not work properly creating new nonexistent syntax errors because the parsing task cannot be interrupted. Furthermore, it is necessary to choose the synchronization point which can force to ignore a part of the input stream or to change it virtually.

The main syntax error recovery strategies are error production, which consists in extending the grammar adding new erroneous productions. Panic error recovery strategy allows the parser to ignore the input stream until find any synchronization token. Finally, the phrase level strategy can make local changes assuming that an unexpected token is correct, for example inserting it into the stack.

Parser generators are commonly used in language processing courses, but the implementation of error recovery methods is far from theory. Obviously, error productions can be used with all parser generators. But, the phrase level is mostly used by LL parser generators, e.g. ANTLR[1]. And the panic mode method is mostly used by LR parser generators, e.g. Yacc[2], Bison[3], and Cup[4]. Even changing the concepts, because the parser developer must specify synchronization points with a special error token, instead of specify synchronization tokens associated with productions.

Visualizing the error recovery process will improve students' understanding of these methods. We have developed VAST [2,4], an educational tool devoted to syntax analysis visualization. VAST has been evaluated in both an educational and observation way [3]. One of the most important features of VAST is its generic approach. It can be used with different parser generators; our students have used it with Cup and ANTLR parser generators. Here we describe how VAST visualizes the different error recovery strategies implemented in both parser generators.

The rest of the article is structured as follows. In the section 2 we describe the related work. In the section 3 we describe the different strategies of error recovery. In the section 4 we explain the visualization of the error recovery with VAST. In the section 5 we describe other characteristics of VAST. Finally, in the section 6 we state our conclusions and future work.

1 http://www.antlr.org/
2 http://dinosaur.compilertools.net/
3 http://www.gnu.org/software/bison/
4 http://www2.cs.tum.edu/projects/cup/

II. RELATED WORK

There exist numerous tools to display some aspects of the compilation process. These tools have been divided according to the use that they are thought for. On one hand there are tools designed to be used mainly with a theatrical aim. However, with this kind of tools the user cannot generate his/her own parsers. Some examples of this type of tools are JFLAP[16], THOTH[8], BURGRAM[9] y SEFALAS[10]. All of them have in common that they animate some aspects of the compilation process such as the table's construction, recognition of the input stream, generation of the syntax tree, etc. On the other hand, there are tools designed to be used with a more practical aim. Some examples of this kind of tools are, ICOMP[6], VisiCLANG[17], APA[12], TREE-VIEWER[19], VCOCO[18], CUPV[11], LISA[14], ANTLRWorks and JACCIE. These tools have two remarkable problems. On one hand they only work with a particular generation tool. On the other hand, they display only certain parts of the compilation process.

There is not any tool that covers the entire parsing algorithm 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 more 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 environments, and to plan their integration in the course. This makes more difficult their use in educational environments [15].

Focusing on both kinds of tools which have been found, and trying to display the error recovery process, we have realized that except ANTLRworks, none of them visualize this process. In the specific case of ANTLRworks, the visualization of the error recovery process is limited. On one hand, it only allows the visualization for parsers built with the parser generator ANTLR, which also mean for LL(1) parsers. Moreover, it only displays a specific kind of error recovery, the ANTLR one, which consists in inserting the unexpected symbol into the parser's stack. Finally, sometimes the visualization generated can be confusing.

In the Figure 1 it is shown the partial ST built by ANTLRworks when a syntax error occurs. In this case the error is produced because the parser expects a symbol which is not in the input stream, so that ANTLR inserts into the stack all the symbols that it founds although there are not correct. In this case as the error cannot be recovered it reaches the end of input stream, finishing the analysis.

III. SYNTAX ERROR RECOVERY, AN OVERVIEW

The main objective of a parser is to build the ST. But if the input stream is erroneous, then the parser must detect as much existing errors as possible. Therefore, the parser cannot stop the analysis after the first syntax error. Instead, it must move to a correct state and continue with the parsing. This is the main idea of SER.

Typically, four SER methods are taught: phrase level recovery, error productions, panic mode and theoretical [1]. Next we briefly describe each method, note that when we qualify a method as simple or complex we are talking about the student's point of view (understanding).

Phrase level recovery is one of the simplest methods. It tries to transform an incorrect phrase into a correct one by inserting/deleting tokens in/from the input stream. Every state of a parser has a list of expected tokens; they can be used in this recovery method. A common example can be found in most C/C++ compilers, the insertion of the forgotten semicolon at the end of sentences.

Error productions detect specific errors by specifying erroneous grammar productions as they would be correct ones. This recovery is directly specified by the language designer/parser developer. The error treatment is defined in the associated actions of the error production.

Panic mode is more complicated because it simultaneously and explicitly, involves the stack, the input stream and the ST.

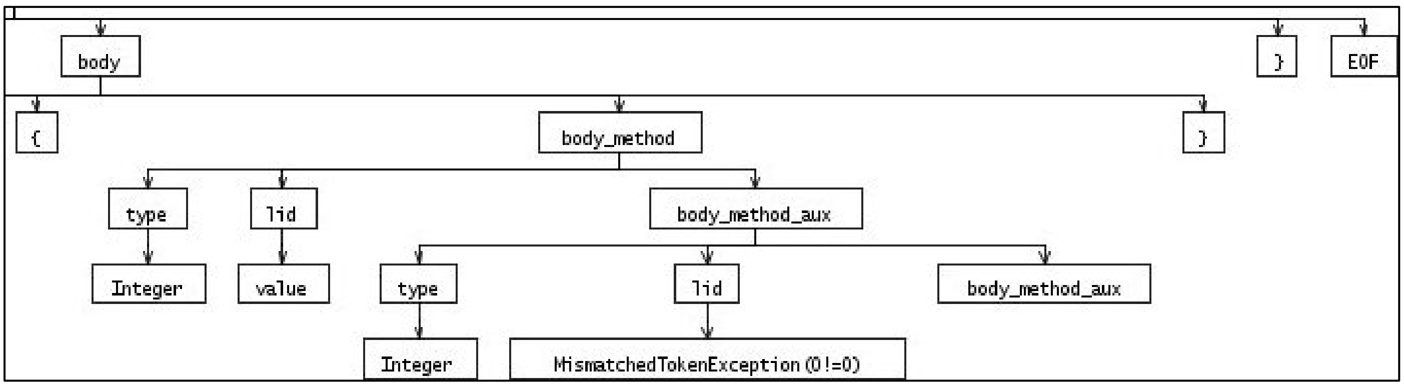


Figure 1. Error recovery visualization in ANTLRworks

This method is based on synchronization points. When an error is detected, the parser discards input tokens until it reaches to a synchronization point. This point is defined by a group of synchronization tokens. To master this method, one must know what are the state of the stack, the input stream and the ST just after the recovery. Besides, it depends on the synchronization token used to recover from the error.

IV. VISUALIZATION OF SYNTAX ERROR RECORY

Visualizing the error recovery process will improve students' understanding of these methods. We have developed VAST [4], an educational tool devoted to syntax analysis visualization. One of the most important features of VAST is its generic approach. It can be used with different parser generators; our students have used it with Cup and ANTLR parser generators. Here we describe how VAST visualizes the different error recovery strategies implemented in both parser generators.

A. *A note about VAST*

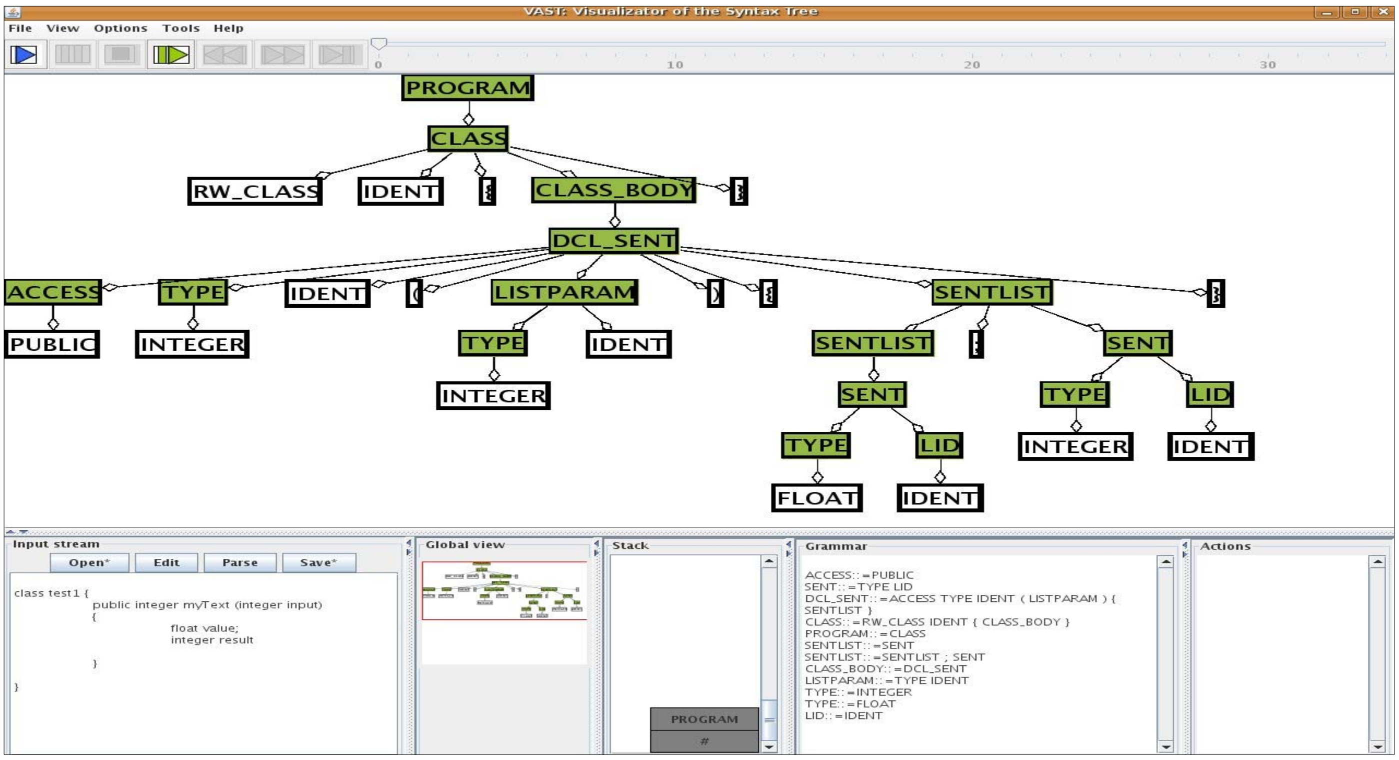
VAST has been designed to cope with any parser generation tool independently from the type of the parser generated (LL/LR). In order to get this independence and keep the easy of use as a fundamental requirement, VAST has been divided in two parts: VASTapi and VASTview.

VASTapi is the part encharged of the language processing, its target is to interpret the actions made by the parsers. Finally, it has to create an intermiddle representation, in a xml file, with the content of the ST and the neccesary information which allows it visualization.

VASTview is the part encharged of the visualization. Its function is to interpret and represent visually the content of the xml created by VASTapi.

In order to make this process work correctly, it is necessary to perform an annotation process. This one consists in inserting calls to the methods of VASTapi using semantic actions inside the parser specification. The information needed by VASTapi is the syntactical rule which has been executed. To make this task, the user has to use the method *addProduction("Antecedent", "Consecuent")* of VASTapi. For each syntactical rule is necessary to add the semantic action which communicates, to VASTapi, the rule which has been used.

In the Figure 2 it is shown the user interface of VAST. In the central part it is displayed the ST and in the bottom the different views of the compilation process (input stream, stack, grammar and actions performed). Besides, it includes a global view to make easier the interaction with the ST. Note that in this figure it is used the horizontal distribution. In the Figure 3 it is shown a scheme of how to work with VAST. The process has been divided in design time, which include the annotation process; execution time, when the parser processes an input stream and as a result of the execution of the VASTapi methods, it is built an intermeddle representation in xml; and finally, visualization time, where VASTview interprets the content of the xml file.



```
class test1 {
        public integer myText (integer input)
        {
                float value;
                integer result

        }

}
```

```
ACCESS::=PUBLIC
SENT::=TYPE LID
DCL_SENT::=ACCESS TYPE IDENT ( LISTPARAM ) {
SENTLIST }
CLASS::=RW_CLASS IDENT { CLASS_BODY }
PROGRAM::=CLASS
SENTLIST::=SENT
SENTLIST::=SENTLIST ; SENT
CLASS_BODY::=DCL_SENT
LISTPARAM::=TYPE IDENT
TYPE::=INTEGER
TYPE::=FLOAT
LID::=IDENT
```

Figure 2. User interface of VAST

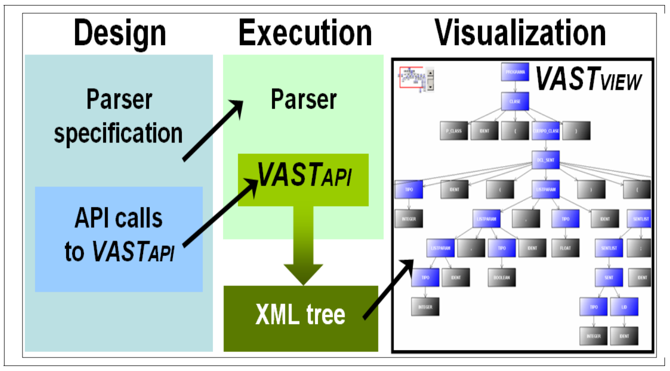


Figure 3. Working with VAST

B. *Visualization of the phrase leve errorl recovery method*

This method is based on token insertion/deletion in/from the input stream. The ANTLR parser generator implements this method by inserting expected tokens. ANTLR allows the parser developer to overload the syntax error recovery method *displayRecognitionError.* We have inserted code that identifies the inserted tokens in the ST.

The main effect of this method is that the terminal nodes of ST (the leaves of the tree) don't correspond to those present in the input stream.

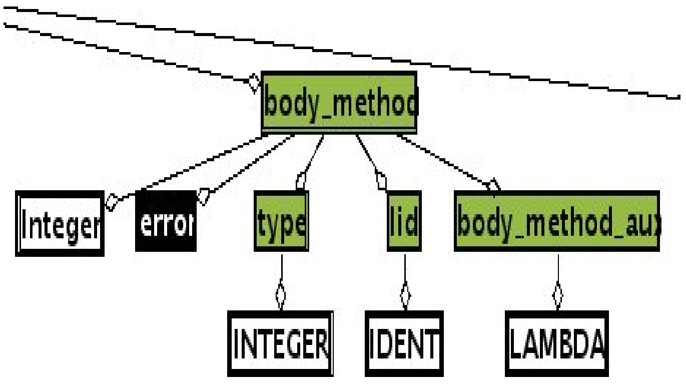
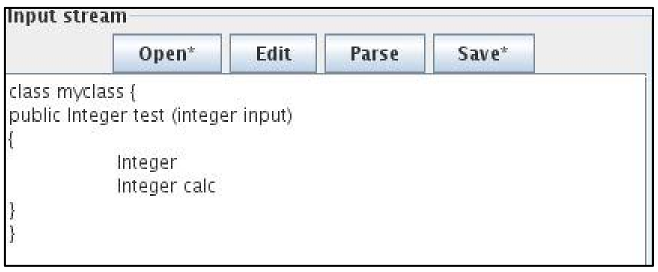


Figure 4. Phrase level recovery with VAST

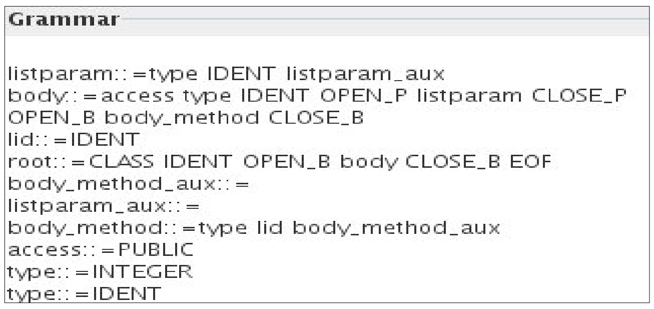
With VAST we highlight the existing tokens of the ST that have been inserted by the phrase level recovery method.



```
class myclass {
public Integer test (integer input)
{
            Integer
            Integer calc
}
}
```

Figure 5. Input stream for LL recovery

In the Figure 4 it is shown an example of an erroneous input stream for a parser generated with ANTLR. As we can see in the input stream in the Figure 5, there is an error in the identifier declaration, so it produces a syntax error. Visualizations can help to understand how this method of error recovery works. When a syntax error occurs, the symbol is shifted into the stack and the analysis continues. In the Figure 6 it is shown the grammar used in the example of the Figure 4.



```
listparam::=type IDENT listparam_aux
body::=access type IDENT OPEN_P listparam CLOSE_P
OPEN_B body_method CLOSE_B
lid::=IDENT
root::=CLASS IDENT OPEN_B body CLOSE_B EOF
body_method_aux::=
listparam_aux::=
body_method::=type lid body_method_aux
access::=PUBLIC
type::=INTEGER
type::=IDENT
```

Figure 6. Grammar for LL parsing

C. *Visualization of error productions*

Taking into account that sometimes the error recovery methods implemented in the generation tools are very generic, using error productions allows to perform a more specific error recovery.

In order to make VAST recognize the error productions, it is necessary to communicate it, which productions are used for this error recovery strategy. To obtain this functionality it has been necessary to add a new method called *addErrorProduction("antecedent", "consecuent")* to VASTapi. As we can see, the information communicated to VAST is exactly the same in both cases, in normal productions and in error ones.

VASTview displays an error production highlighting the whole production in a different colour. As result, the user can distinguish that a specific error recovery has been performed.

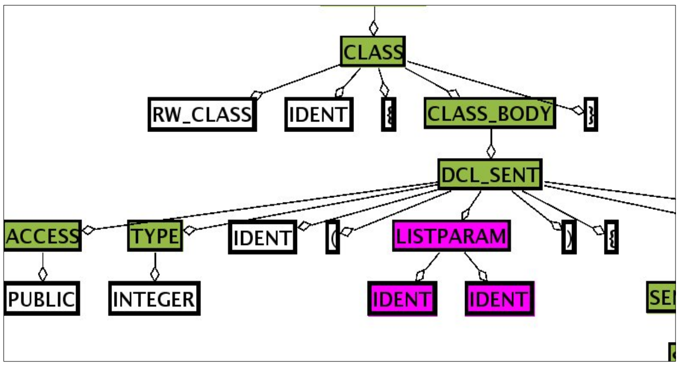
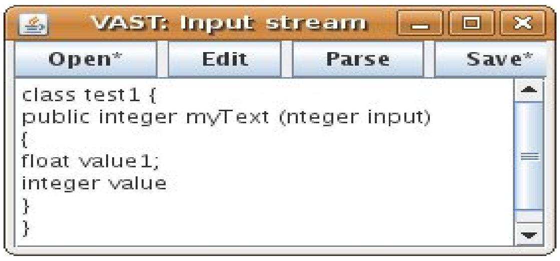


Figure 7. Visualization of error production

In the Figure 7 it is shown an example of error recovery using the error production method. In this case the parser recovers from a specific error produced by an erroneous input stream in the parameters declaration of a method. In the Figure 8 it is shown the input stream using for this example. We can see an error in the parameters declaration of the method *myText.* In the Figure 9 it is shown a fragment of the annotation of the Cup specification to implement this behaviour using error productions. Note that we are using an error strategy not implemented by the generation tools. However, VAST can distinguish this kind of productions, so the visualization adopts the correct aspect.

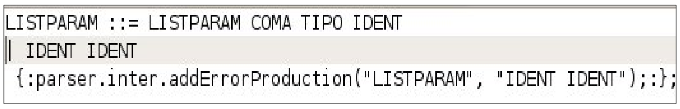


```
class test1 {
public integer myText (nteger input)
{
float value1;
integer value
}
}
```

Figure 8. Input stream for error production recovery

D. *Visualization of panic mode method*

Normally, the panic error recovery is implemented in the generation tools using a special symbol, usually called "error" symbol. This recovery method allows to define easily the synchronization points, however, sometimes it can be extremmely complex to understand how the error recovery strategy works[1].

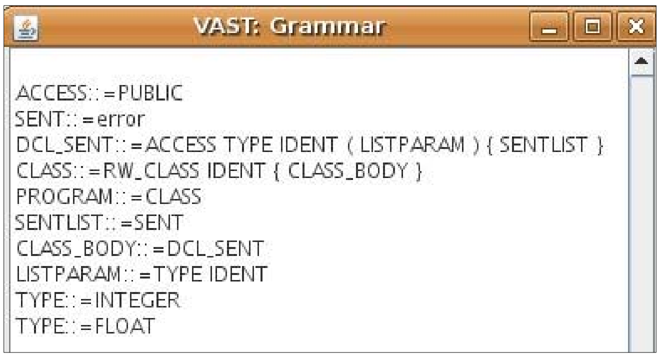


```
LISTPARAM ::= LISTPARAM COMA TIPO IDENT
| IDENT IDENT
 {:parser.inter.addErrorProduction("LISTPARAM", "IDENT IDENT");:};
```

Figure 9. Annotation for error production recovery

In the specific case of the Cup generator, the panic error recovery is implemented using the Terminal symbol "error". When the parser detects an error in the input stream, it commutes to an internal error stage, inserting into the stack the "error" symbol. To extract this one from the stack it is necessary to find a production which allows to shift the error. If the error can be shifted, then the parser has recovered correctly and the analysis continues. However, if the parser reaches to the grammar axiom and the error has not been shifted, it has not recovered from the error and the analysis will conclude.

To display correctly this process in VAST, it was necessary to implement new functionalities in VASTapi for ascendant parsers. There exist two problems when trying to visualize this process. On one hand, when the parser enters in the error mode, it does not perform any reduction, so that VASTapi does not receive any information of the task done by the parser. On the other hand, as the parser is in an error mode, it does not communicate the ignored symbols to VASTapi. Due to these problems and to obtain these information, it was necessary to modified VASTapi.



```
ACCESS::=PUBLIC
SENT::=error
DCL_SENT::=ACCESS TYPE IDENT ( LISTPARAM ) { SENTLIST }
CLASS::=RW_CLASS IDENT { CLASS_BODY }
PROGRAM::=CLASS
SENTLIST::=SENT
CLASS_BODY::=DCL_SENT
LISTPARAM::=TYPE IDENT
TYPE::=INTEGER
TYPE::=FLOAT
```

Figure 10. Grammar used for panic error recovery

Instead of solving these problems independently, we have chosen a global solution, which allows to solve both of them. When the parser enters in the error mode, it calls to the *syntax_error* method, which has been overload to communicate to VASTapi that the parser is in the error mode. Besides, in this context, although the parser is in the error mode, it continues asking for symbols to the lexer using the *scan* method.

When VASTapi receives the symbols from the lexer, it marks them indicating their mode (normal or error). Finally, when the error production is performed and the "error" symbol detected, VAST has all the necessary information. On one hand there are the subtrees processed. On the other hand it is the ignored input stream. With this information VAST can create the visualization of this error recovery method.

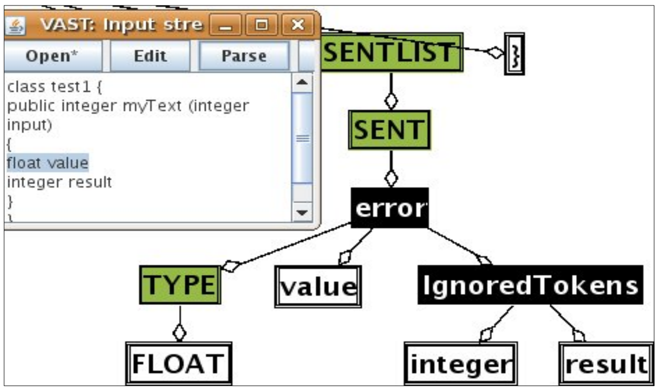


Figure 11. Visualization of the panic error recovery with 1 error

To this end, firstly it is necessary to obtain the synchronization symbol. This one shows the point in which the parser continues the analysis without ignoring the input stream. We obtain this symbol acceding to the last one received when an error production is applied. Secondly, to obtain the ignored symbols it is necessary to access to the received symbols since the parser commuted to the error mode until it received the synchronization symbol. With this information VAST builds a subtree, which root node is the error symbol. The sons of this node are the subtrees processed correctly just before the error occurred and an additional node with the ignored symbols

("IgnoredTokens" node), which contains each one of the ignored symbols.

In the Figure 11 it is shown an example of the panic error recovery. In this case there exists only an error in the input stream, produced because the symbol";" is not after the identifier "*value*". The error recovery implemented used the symbols ";" y "}" as synchronization symbols, so that the parser ignores the input stream until it finds one of those symbols. In this example the parser ignores the "*integer*" and "*result*" symbols. When the parser reads the symbol "}", it recovers from the error and continues the analysis.

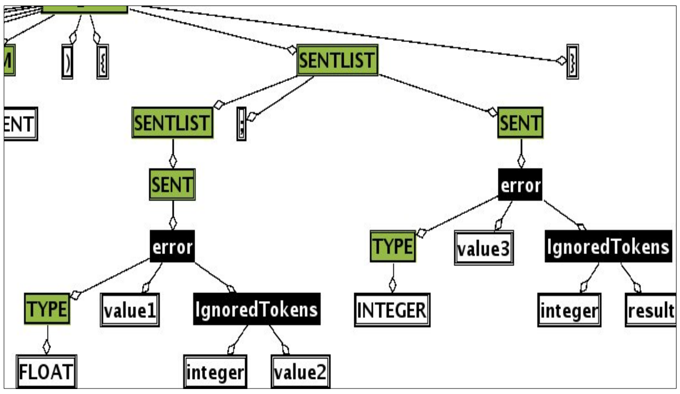
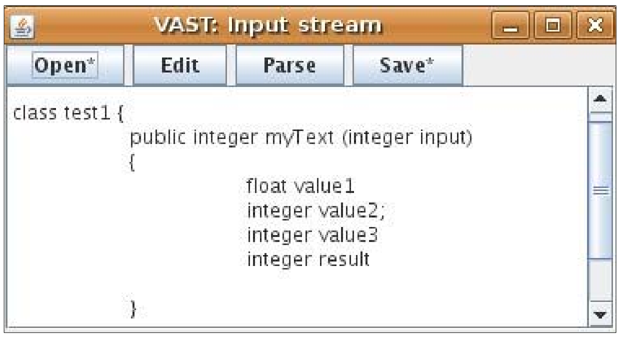


Figure 12. Visualization of the panic error recovery with 2 errors

In the Figure 12 it is shown an example of panic error recovery in which the parser recovers from two errors in the input stream, both produced because there is not ";". As in the previous example, the parser implemented uses the ";" and "}" as synchronization symbols. The first error is produced because after the symbol "value1" it is expected the ";" symbol. However, the parser reads from the input stream the "integer" symbol, so that it enters in the error mode, from which it exits only when reads the ";" symbol. In this case the ignored symbols from the input stream are "integer" and "value2". The second error is produced because as in the first case, there is not ";" symbol. In this case the parser ignores the "integer" and "result" symbols from the input stream. In the Figure 13 it is shown the input stream used to generate the tree of the Figure 12.



```
class test1 {
        public integer myText (integer input)
        {
                float value1
                integer value2;
                integer value3
                integer result

        }
```
Figure 13. Input stream for panic error recovery with 2 errors

In the Figure 14 it is shown an example in which there are two consecutive errors. In this case as there is not synchronization symbol between both errors, the parser ignores more symbols from the input stream. In the Figure 15 it is shown the input stream used to generate the visualization of the Figure 14.

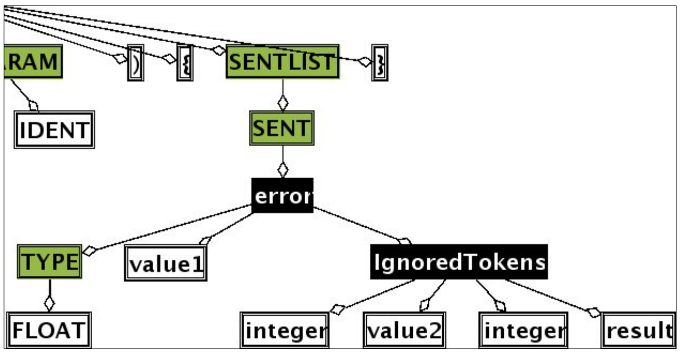
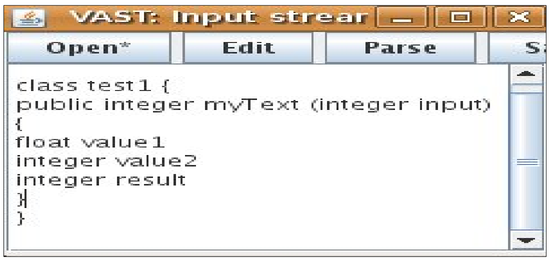


Figure 14. Visualization of panic error recovery with 2 errors consecutive

V.OTHER CHARACTERISTICS OF VAST

Visual representations are not effective educational resources by default, they must be carefully designed. VAST has passed a number of evaluations. As a consequence, we have designed VAST to provide a number of useful features.



```
class test1 {
public integer myText (integer input)
{
float value1
integer value2
integer result
}
}
```
Figure 15. Input stream for error panic recovery with 2 errors consecutive

Apart from inserting additional functionality in VASTapi to interpret the error recovery, we have modified VASTview to handle new functions. It has been improved those characteristic which allow to distribute VAST wider and easier. To this, the user interface has been internationalized into Spanish and English. Besides, we have worked in the platform independence, allowing to work with VAST in Windows, Linux and Mac OS systems. Moreover it has been performed some improvements to work easily with the tool and also to display some specific aspects of the error recovery process that cannot be treated by VASTapi.

A. *Visualization of the ST*

The graphical representation which has been chosen to visualize the ST is the tree structure resulting of the processing of the input stream. In previous versions of VAST, it can be distinguished three types of nodes: terminal nodes (T), non terminal nodes (NT) and error nodes (EN).

The T nodes are the leaves of the tree, the NT are the internal nodes of the ST and the EN represent the synchronization points in which the parser can recover from an syntactical error. The last one allows to determine the exact place where it is recovered (if possible) a syntactical error. Furthermore, they allow to guess the amount of recognized elements, so the elements recognized before an error occurs will be sons of the corresponding error node.

Due to the changes performed to improve the visualization of the error recovery in panic mode, it has been necessary to create an additional type of nodes, ignored nodes (IN). This kind of nodes has only sense when it is implemented a panic error recovery strategy. When the parser detects an error, it ignores the input stream until it finds a synchronization symbol. The function of the IN is to group the ignored symbols until a syntactical error is recovered. In the graphical representation they appear as sons of the EN, containing all the T ignored.

B. *Reproduction of the construction of the ST*

The animation of the construction process of the ST is performed using different intermeddle stages generated and ordered by VASTapi. The animation of this process helps the students to understand how the input stream is processed using shifts and reductions.

To reproduce an animation, VASTview has a VCR controls and a slide bar which allows to reach easily to a specific point of the reproduction. During the construction process the ST change its shape, area and content. Due to this behaviour the interface could adapt to each stage using a “best-fit” policy, which would change the location of the nodes of the tree and consequently affect the user. According to this, we have decided to keep the location of the nodes of the tree as the construction process of the ST is reproduced.

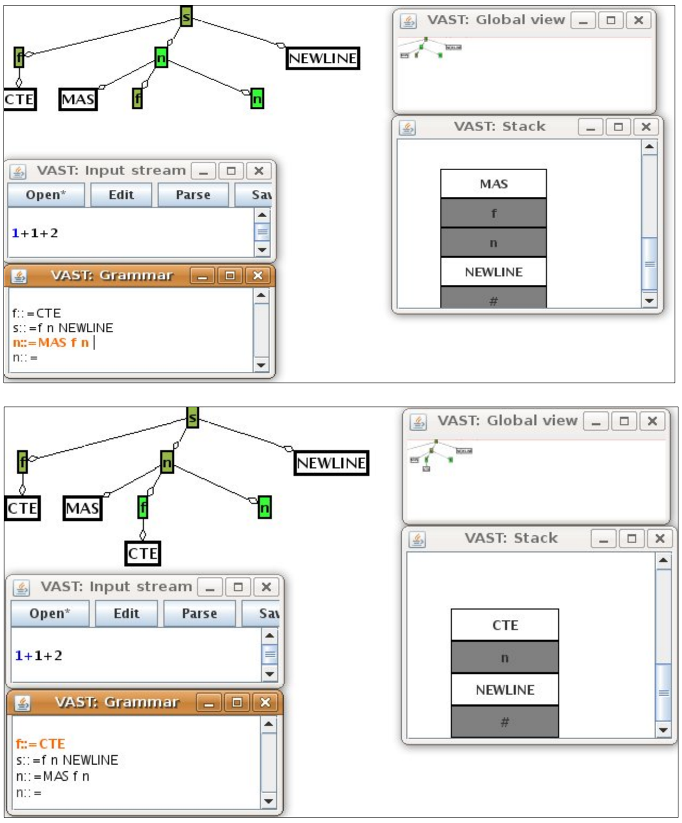
The reproduction process is synchronized with all the views offered in VAST, so that when the state of the ST changes all of the views have to act consequently.

In the Figure 16 it is shown an example of the reproduction process of the construction of the ST. In this case the parser is descendant, so as VAST constructs the ST the current node is remarked. In this figure we can see two intermeddle stages of the reproduction.

C. *Different distributions*

Probably the ST generated by the parsers designed by the students are huge and without any specific structure (symmetric, width and height). For this reason, VASTview has an user interface which make easier the visualization and interaction of the ST.

The interface of VASTview offers three different distributions: horizontal, vertical and float. The horizontal and vertical distributions are used for horizontal and vertical ST respectively. In the case that a ST cannot be classified in any of these types, VASTview offers the float distribution in which the user can change the position of the different views.



```
1+1+2
```

```
f::=CTE
s::=f n NEWLINE
n::=MAS f n
n::=
```

```
1+1+2
```

```
f::=CTE
s::=f n NEWLINE
n::=MAS f n
n::=
```

Figure 16. Reproduction of the construction process for LL parser

Independently of the distribution used, the interface of VASTview has a global and detail view of the ST, all together with the zoom and aggregation functionalities. The global and detail view allow the user to manipulate easily the ST. The global view indicates the visible part of the ST in the detail view. The functionalities of the global view give the students the opportunity of examining the ST with zoom, and aggregation, keeping always the synchronization with the global view.

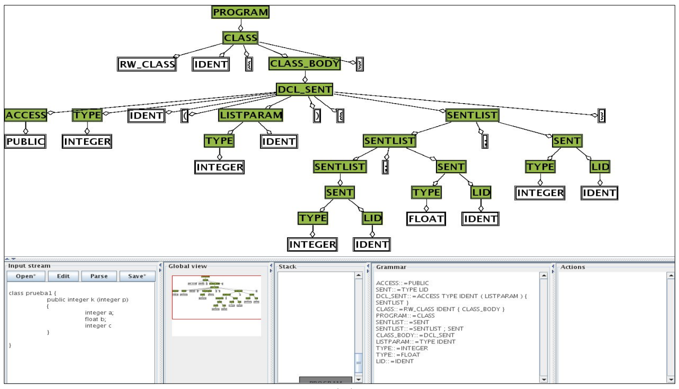
The aggregation allows the user to hide those parts of the ST which are not interesting. In this case, when a node is resumed, we used a “best-fit” policy to redistribute the ST.

In the Figure 17 it is shown the different distribution of the VAST interface. Figures 17a, 17b and 17c show the horizontal, vertical and float distribution of the user interface respectively.

D. *Views of the compilation process*

Apart from displaying the ST, VASTview visualizes the input stream, the stack of the parser, the used grammar and textual explications of the performed actions. Each of these views have a different aim, however, all of them are really important when the construction process of the ST is animated.

In case of the input stream, it is remarked when the construction process is reproduced. When in the parser is

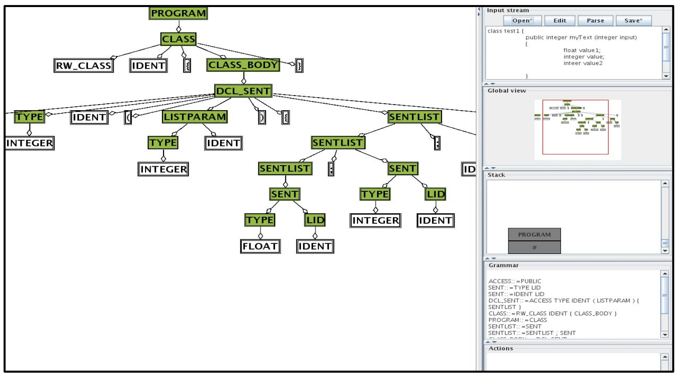


```
class prueba1 {
        public integer k (integer p)
        {
                integer a;
                float b;
                integer c
        }

}
```

```
ACCESS::=PUBLIC
SENT::=TYPE LID
DCL_SENT::=ACCESS TYPE IDENT ( LISTPARAM ) {
SENTLIST }
CLASS::=RW_CLASS IDENT { CLASS_BODY }
PROGRAM::=CLASS
SENTLIST::=SENT
SENTLIST::=SENTLIST ; SENT
CLASS_BODY::=DCL_SENT
LISTPARAM::=TYPE IDENT
TYPE::=INTEGER
TYPE::=FLOAT
LID::=IDENT
```

(a)

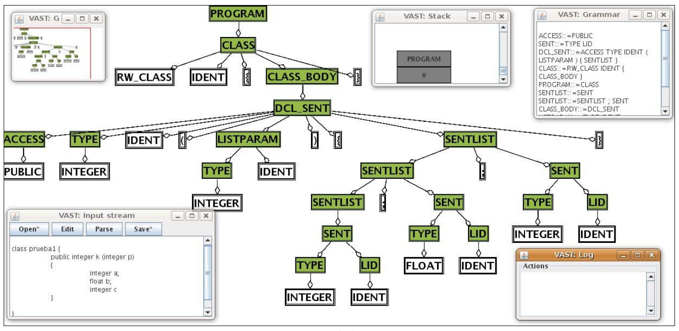


```
class test1 {
        public integer myText (integer input)
        {
                float value1;
                integer value;
                inteer value2

        }
```

```
ACCESS::=PUBLIC
SENT::=TYPE LID
SENT::=IDENT LID
DCL_SENT::=ACCESS TYPE IDENT ( LISTPARAM ) {
SENTLIST }
CLASS::=RW_CLASS IDENT { CLASS_BODY }
PROGRAM::=CLASS
SENTLIST::=SENT
SENTLIST::=SENTLIST ; SENT
```

(b)



```
ACCESS::=PUBLIC
SENT::=TYPE LID
DCL_SENT::=ACCESS TYPE IDENT (
LISTPARAM ) { SENTLIST }
CLASS::=RW_CLASS IDENT {
CLASS_BODY }
PROGRAM::=CLASS
SENTLIST::=SENT
SENTLIST::=SENTLIST ; SENT
CLASS_BODY::=DCL_SENT
```

```
class prueba1 {
        public integer k (integer p)
        {
                integer a;
                float b;
                integer c
        }
}
```

(c)

Figure 17. VAST interface distributions

implemented a method of error recovery, the input stream can distinguishes between the process symbols and the ignore ones. Besides, when the ST is displayed in a static way, if a T node is selected, then in the input stream is remarked the part which corresponds with that node.

The stack view simulates the content of the stack of the parser. It is compatible with both LL/LR parsers and it is really important in the error recovery process, so it is possible to make predictions about the recovery point just observing the content of the stack and the grammar. This view allows to display a stack history, which allow the user to display the different stages of the stack during the analysis.

The visualization of the grammar allows to indicate which syntactical rule has been used in a reduction/derivation. In this case it is necessary to distinguish between the grammar of the specification and the grammar used. The grammar of the specification refers to the set of rules that implements a parser. However, the grammar used is a subset of the previous one, so it is the part of the specification grammar used to process a specific input stream. According to the way of work of VASTapi, it is easier to get the used grammar. So that, when it is displayed the set of rules, it is important to have into account that it refers to the used grammar.

The textual explication was a characteristic asked by the students during the evaluation sessions of VAST. In this case, it has been added the explications of the actions performed during the construction of the ST, including the error recovery process.

Finally, it has been improved the way of showing the views of the VASTview. In this version is possible to hide or change the workspace of each different view. This solution has been necessary to work with low resolution screens.

E. Importing parsers

According to the results of the usability evaluation of VAST [5], it was necessary to implement a global integration which consists of two functionality integrations.

The first integration allows to anote a syntactical specification, generate and compile it. The second one allows to execute a parser and visualize the result of this execution.

As result of the implementation of VAST, it has been developed a first approximation of the importation of parsers. In the case of the second integration it has been developed in a generic way. However, the first integration has been implemented just for the ANTLR specification. This limitation does not involve VAST, but the necessity of creating metaparsers which can annote automatically the specification including the VASTapi calls.

VI. CONCLUSIONS

In this work, we has presented the educational tool VAST, which allows to visualize the ST and the different views of the compiling process. The main characteristics of VAST are the independence from the generation tool and the easy of use.

Specifically, within the visualization of the ST, we have studied the existing tools and realized that the existing tools to display the compilation process, give a partial or/and particular solution. On one hand, these tools are particular because they only allow to display the parser created for a concrete generation tool. On the other hand, they give a partial solution because they do not show all the views of the compilation process.

In this context when we focus on the syntax error recovery visualization, we observe that none of these tools give a solution for it. We has found only one tool, ANTLRworks, which give an approximation of the syntax error recovery visualization. However, as in the other cases, the solution implemented is particular and partial. Taking into account that SER is one of the most difficult points in the language processor courses, and the pedagogical effect that visualizations have in education, we have decided to use the visualizations technologies to improve the comprehension process of the syntax error recovery.

We have added new functionalities to VAST, in order to display the syntax error recovery. We have detailed the different characteristics of each error recovery strategy. Afterwards, we have explained how they are implemented in VAST. As a result VAST allows the visualization of the insertion error recovery, error productions and panic mode error recovery.

Furthermore, in this work we include the functionality of importing parsers specifications. This one allows to display any parser in a very easy way.

The work with VAST has not finished yet. As future works we plan different evaluations of the new functionalities. Moreover, we will work on the exporting of animations to be used in different contexts (e.g Powerpoint). Finally, it will be necessary to improve the importation of parsers, so this version allows only to import automatically ANTLR specifications.

VII.ACKNOWLEDGEMENTS

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

REFERENCES

[1] A. V. Aho, M. S. Lam, R. Sethi and J. D. Ullman, Compilers:Principles, Techniques, and Tools, Prentice Hall, 2007.

[2] F. J. Almeida-Martínez, Jaime Urquiza-Fuentes, and J.Ángel Velázquez-Iturbide. VAST: Visualization of Abstract Syntax Tree within Language Processors Courses. In SoftVis '08: Proceedings of the 4th ACM Symposium on Software Visualization, pages 209–210, New York, NY, USA, 2008. ACM.

[3] F. J. Almeida-Martínez and Jaime Urquiza-Fuentes. Syntax Trees Visualization in Language Processing Courses. In Proceedings of the Nineth IEEE International Conference on Advanced Learning Technologies, 2009. ICALT 2009., page 597-601, Los Alamitos, USA, 2009. IEEE Computer Society Press.

[4] F. J. Almeida-Martínez, J. Urquiza-Fuentes and J.Á. Velázquez-Iturbide, "Visualization of Syntax Trees for Language Processing Courses" Journal of Universal Computer Science vol. 15(7), pp 1546-1561, 2009.

[5] F. J. Almeida-Martínez and J. Urquiza-Fuentes. Teaching LL(1) parsers with vast. An usability evaluation. Technical report, http://www.dlsi1.etsii.urjc.es/doc/DLSI1-URJC_2009-01.pdf, 2009.

[6] K. Andrews, R. R. Henry, and W. K. Yamamoto. Design and implementation of the UW illustrated compiler. In Programming Language Design and Implementation '88: Proceedings of the ACM Special Interest Group on Programming Languages 1988 conference on Programming Language Design and Implementation, pages 105–114, New York, NY, USA, 1988. ACM.

[7] S. K. Card, J. D. Mackinlay and B. Shneiderman, Readings in information visualization, using vision to think, Academic Press, 1999.

[8] César García Osorio, Mediavilla Sáiz, Iñigo I Javier Jimeno Visitación y Nicolás García Pedrajas. Teaching Push-Down Automata and Turing Machines. In Innovation and Technology in Computer Science Education '08: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, pages 316–316, New York, NY, USA, 2008. ACM.

[9] César García-Osorio, Carlos Gómez-Palacios, and Nicolás García-Pedrajas. A Tool for Teaching LL and LR Parsing Algorithms. In Innovation and Technology in Computer Science Education '08: Proceedings of the 13th Annual Conference on Innovation and Technology in Computer Science Education, pages 317–317, New York, NY, USA, 2008. ACM.

[10] J-F Jodar-Reyes and J. Revelles-Moreno. SEFALAS: Software para la Enseñanza de las Fases de Análisis Léxico y Análisis Sintáctico.

[11] Alan Kaplan and Denise Shoup. CUPV. A Visualization Tool for Generated Parsers. In Special Interest Group on Computer Science Education '00: Proceedings of the thirty-first Special Interest Group on Computer Science Education Technical Symposium on Computer Science Education, pages 11–15, New York, NY, USA, 2000. ACM.

[12] Sami Khuri and Yanti Sugono. Animating parsing Algorithms. In Special Interest Group on Computer Science Education '98: Proceedings of the twenty-ninth Special Interest Group on Computer Science Education Technical Symposium on Computer Science Education, pages 232–236, New York, NY, USA, 1998. ACM.

[13] J. R. Levine, T. Mason and D. Brown, Lex & yacc, 3nd ed., O'reilly, 1995.

[14] M Mernik and V Zumer. An educational tool for teaching compiler construction. 46(1):61–68, 2003.

[15] Naps, T., Rößling, G., Almstrum, V., Dann, W., Fleischer, R., Hundhausen, C., Korhonen, A., Malmi, L., McNally, M., Rodger, S. and Velázquez-Iturbide, J.: "Iticse 2002 working group report: Exploring the role of visualization and engagement in computer science education"; Special Interest Group on Computer Science Education Bull. 35, 2 (June 2002), 131-152.

[16] Stephen Reading Susan H. Rodger, Jinghui Lim. Increasing interaction and support in the formal languages and Automata theory course. Innovation and Technology in Computer Science Education pages 19–24, 2007.

[17] D. Resler. VisiCLANG. A Visible Compiler for CLANG. Special Interest Group on Programming Languages Not., 25(8):120–123, 1990

[18] R. Daniel Resler and Dean M. Deaver. VCOCO: A Visualisation Tool for Teaching Compilers. In Innovation and Technology in Computer Science Education '98: Proceedings of the 6th annual Conference on the Teaching of Computing and the 3rd annual Conference on Integrating Technology into Computer Science Education, pages 199–202, New York, NY, USA, 1998. ACM.

[19] Steven R. Vegdahl. Using Visualization Tools to Teach Compiler Design. In Proceedings of the Second Annual Consortium for Computing Sciences in Colleges on Computing in Small Colleges Northwestern Conference, pages 72–83,USA, 2000. Consortium for Computing Sciences in Colleges.