# Designing and developing software for educative virtual laboratories with language processing techniques: lessons learned in practical experiments

José Jesús Castro-Schez

Escuela Superior de Informática
Paseo de la Universidad, 4
13071 Ciudad Real, España
JoseJesus.Castro@uclm.es

Miguel A. Redondo

Escuela Superior de Informática
Paseo de la Universidad, 4
13071 Ciudad Real, España
Miguel.Redondo@uclm.es

Jesús Gallardo

Escuela Universitaria Politécnica
Ciudad Escolar, s/n
44003 Teruel, España
Jesus.Gallardo@unizar.es

Francisco Jurado

Escuela Superior de Informática
Paseo de la Universidad, 4
13071 Ciudad Real, España
Francisco.Jurado@uclm.es

*Systems that facilitate automation in the evaluation of learning tasks are usually domain dependent. Furthermore, they usually do not allow users to specify new tasks which are described by the users themselves. Therefore, the development of activities in which work is done in the higher levels of Bloom's Taxonomy is limited. In this article, we introduce a proposal made up by a series of guidelines to develop educational software that allow users to specify their own learning activities, to add alternative solutions and to receive an evaluation from the system so that they are guided in their learning process. This proposal is based on the use of formal languages to specify activities and their solutions, and also on the use of language processors for such languages, so that it is possible to build a computable model that solves the activities and analyzes the solutions. The proposal is consequence of the lessons learned from a real experience, which is the Proletool application, a tool designed for teaching syntax analysis techniques in the subject Language Processors.*

*Categories and Subject Descriptors: D.2.2 (Software Engineering): Design Tools and Techniques; H.1.2 (Models and Principles) User/Machine Systems; H.5.2 (Information Interfaces and Presentation) User Interfaces; K.3.1 (Computers and Education) Computer Uses in Education*

## 1. INTRODUCTION

Information technologies and, in particular, computer applications and systems represent a basic instrument for carrying out learning activities, allowing learners to work in the higher cognitive levels while acquiring new knowledge and developing competences and abilities (Churches, 2008). The work described in this paper is centered in the particular context of systems that assist students in the process of building their own knowledge by providing feedback regarding the consequences of actions carried out during the course of a learning activity. In such systems, the basis for the building of knowledge is the information provided to the students as feedback regarding what they are doing right and what they are doing wrong while carrying out a learning activity. This helps them to reevaluate, improve their decisions and consider different options (Bravo, Van Joolingen and De Jong, 2009; Gordijn and Nijhof, 2002; Kumar, 2004; Sanders and Hartman, 1987).

The purpose of this work is twofold. On one hand, we want to show how the *Proletool* application was designed. *Proletool* is a teaching tool that is used to improve the processes of teaching and learning in the subject *Language Processors*, by working in the higher cognitive levels of the *Bloom's Taxonomy* (Bloom, 1965). On the other hand, our aim is to present the conclusions obtained and the lessons learned during the development of the aforementioned tool and of another teaching tool called *Selfa* (Castro-Schez et al., 2009), so that we can establish some guidelines that may be used in the design and development of e-learning systems in other fields by using the ideas underlying such tools. Furthermore, a practical example of the application of these guidelines to a different field of formal languages is shown.

The learning objectives in any teaching technique imply that students should be able to reach the higher cognitive levels while they acquire new knowledge and develop competences and abilities (Anderson and Krathwohl, 2001; Bloom, 1965). In order to achieve this, teachers can use several methods for teaching/learning (Oser and Baeriswyl, 2001). In education, and specifically in the fields of science and engineering, Problem-Based Learning is one of the principal methods employed (Dewey, 1933), where the objective is to promote active, student-centered learning and to combine theoretical lectures and laboratory learning activities (Bonwell and Eison, 1991; Eden et al., 1996; Makkonnen, 2000; Mcconnell, 1996).

In this sense, the aim of our proposal is to facilitate, in a systematic way, the possibility for learners to formulate new problems (or substantially change existing ones), as well as providing their solutions. Our proposal also considers the fact that the computer system that supports the resolution of such problems should provide information (feedback) regarding the accuracy of the solutions provided. Thus, as many alternatives and variants as desired may be checked, and students may add their own mechanisms for validation. In short, in the context of problem solving, learning activities in which work is done in the application, synthesis and evaluation levels of

Bloom's Taxonomy may be promoted.

Furthermore, feedback provided by the tool can be useful not only for students, but also for its inclusion in tools employed in automatic advice generation (Barros, et al., 2005; Bravo, Van Joolingen and De Jong, 2009; Chatley and Timbul, 2005; Joy, Griffiths and Boyatt, 2005; Jurado et al., 2009; Perez, Paule and Cueva, 2006; Truong, Roe and Bancroft, 2005). Thus, other applications will be able to use this information to evaluate the degree to which learning objectives are achieved, to decide about the need for planning new learning activities, and to adapt the learning process to the cognitive characteristics of the students (Jurado et al., 2008).

The rest of the paper is structured as follows: firstly, the *Proletool* application will be introduced (Section 2); secondly, we will present our proposal of guidelines for the design of learning tools that support the specification and evaluation of problems and solutions (Section 3); subsequently, some issues related to the application that has been developed will be discussed in order to check the suitability of the proposal (Section 4); followed immediately by a commentary on the results obtained as a consequence of the use of the concept in the field of Language Processors; and finally, some remarks by way of conclusion will be compiled (Section 5).

## 2. *PROLETOOL*: LEARNING THE *LANGUAGE PROCESSORS* SUBJECT

Hereafter, we will explain how the design and implementation of the *Proletool* application (available at http://portal.esi.uclm.es/proletool/) was carried out. Also, some guidelines for the design of learning tools that work on the higher levels of Bloom's Taxonomy based on our experience are defined. In the following sub-sections, the application will initially be put in context in order to highlight the general requirements of the tool. Subsequently, we will focus on the design process that has been followed and, finally we will show some results.

### 2.1. Framework of Application of *Proletool*

*Proletool* is intended to assist students in the process of teaching/learning in the subject *Language Processors*, as well as to help teachers convey instruction or, indeed, to allow teachers delegate some work to the tool. The main goal of the *Language Processors* profile is to introduce the student to the problems that arise when a language processor, translator, compiler or interpreter of a computer language is designed, and also to the application of techniques in solving such problems.

For the subjects included in this profile, some knowledge and ability with respect to grammars and formal languages is required to allow for an understanding of, and an ability to contend with the process of solving problems that arise in the analysis and design of language processors. Usually, the subject content is structured in six thematic units that consist of one, or several lessons each. These units are determined by the stages in the design and development of an application of this kind: introduction to translators, lexical analysis, syntactic analysis, semantic analysis and code generation, execution environment, and synthesis stage.

When a language processor is being built in this domain, the main issue to be dealt with is the design and development of a syntax analyzer or *parser*. Therefore, the most important aspect of language processors is the syntactical analysis or *parsing*, requiring the detailed study of methods of top-down parsing (e.g., LL1) as well as bottom-up parsing (e.g., SLR1, LR1 and LALR1).

In order to learn these methods in a sound and efficient way, it is important, not only to acquire the skill that is needed for their application, but also to perform and correct a great number of practical exercises. However, this usually cannot be done in groups during face-to-face classes, due to the extent of the syllabus and to the limited time of the classes. To solve such restrictions we have two options: the first one consists in providing students with a list of exercises together with their solutions, so that they can assimilate and learn parsing techniques in an individual and autonomous way. The second one, which is the option proposed in this paper, consists of delegating that task to a software tool, in this case, the *Proletool* application.

This tool is intended to support the learning of parsing techniques of the LL1, SLR1, LR1 and LALR1 types, allowing students to suggest new exercises and their respective solutions. Later, the tool corrects those exercises and, if the solutions are not the correct ones, it shows where mistakes have been made, and what can be done to avoid a repetition of those mistakes.

### 2.2. Requirements Analysis

In the design of *Proletool*, the services offered by a series of tools for teaching and studying parsing techniques currently in existence were taken into account (e.g., Caburé! in the SEPa[1] Project, JFLAP[2], SEFALAS[3], ANAGRA[4], JavaPars[5]). After a detailed study of each tool, the following conclusions were reached with respect to what functionality *Proletool* should add:

- It should be a web-based application in order to make it easier to use, as local installation is not needed.
- It should allow the preparation of any number of exercises off-line, that is, without being connected to the tool.
- It should accept exercises from the students relating to the application of parsing techniques to grammars, build solutions to such exercises and make corrections using the information generated to guide students in their learning process.
- It should allow the simulation of a syntactic analysis for a given input in any exercise that has been introduced, showing how the generated solutions are used in that simulation process.
- The user interface of the application should allow access to all the important parts of the application. Also, it should allow interaction with the contents of the solutions. and navigation between sections of the application should be as easy as possible.
- The application should facilitate monitoring of the activity being carried out by each student. This information allows the teacher to control the learning process being followed by the students.

---

1 http://www.ucse.edu.ar/fma/sepa
2 http://www.jflap.org
3 http://lsi.ugr.es/~pl/software.php
4 http://webdiis.unizar.es/~ezpeleta/COMP I/compiladoresI.htm
5 http://paginaspersonales.deusto.es/josuka/jparser/parser.html

- Information should be presented graphically in order to make it more understandable. For instance, automata that are used in bottom-up parsers (SLR1, LR1 and LALR1).

*Proletool* also includes general support for functionality such as: management and administration of users and permissions; publication of news related to the tool itself; logging of usage statistics and generation of textual and graphical reports; help for users of the tool in answering questions about its use and about how it works; compilation of comments and suggestions regarding improvement; and management and administration of a collection of examples that are useful in explaining methods, and in guiding the learning process of the student.

### 2.3. Main Design Decisions

The first important design decision that should be considered refers to the definition of a communication mechanism rich enough to allow students to define, in a precise way, the problems they want solved by the tool, specifying the steps that have been followed or the problems and solutions to be corrected. In selecting such a communication mechanism, some alternatives used nowadays have been considered: forms (Cole, Wainwright and Schoenefeld, 1998), diagrams or graphics (Rodger and Finley, 2011; Tamagnini et al, 2011), and input languages (Sierra et al, 2008; Castro-Schez et al, 2009). The main advantage of communication by means of forms is the fact that a guided process is provided, so that at every moment during the process, the student is shown what to do. This makes it quite suitable for students that are unable to either create or formalize problems, or to define solutions for them, given that they are limited to answering predefined questions. However, when forms are used, interaction with the tool is slow, and the option of off-line work is not supported. The option of diagrams or graphics is also interesting for students without a great knowledge in the area, but this option suffers from the same disadvantages as form input. Furthermore, neither of these two mechanisms allows intermediate results to be stored or further experimented with.

Due to the problems that constrain the previous alternatives, the communication mechanism that was chosen in *Proletool* is a formal Domain-Specific Language (DSL) that allows writing exercises (i.e. grammar and parsing technique) and solutions (parsing tables). Although it forces the student to learn a language (its vocabulary, its syntax and the meaning of its constructions), the main characteristic of this option is that it forces students to think and to create problems, as well as to propose and specify solutions to those problems in an easy and systematic way. This mechanism encompasses the main objective that characterizes the technique that we are proposing. However, the main criticism is that time to study a language can be quite a challenge. In response to this, we point out that authoring tools could be used to help the student to write exercises in the DSL of the tool. We highlight that a DSL is a computational language and it should be easy to build an authoring tool, for example, in order to use visual representations.

The next important decision to be taken relates to providing the system with the capacity to understand the input supplied by the student (problems and solutions), to acquire the information needed to solve the problem, and to explain the solutions that may be reached. This entire process should be done in a systematized and automated way. Furthermore, solutions that are generated can be used to check if a given

solution proposed by a student is correct or not, that is, if it matches any of the solutions generated by the system. Considering the results of this evaluation process, some learning activities can be proposed to the student (e.g., a recommendation of problems from a given set, a suggestion to review a given lesson, etc.). To provide the system with such ability, *Proletool* must have a processor for such a language with the capacity to check if the input text has the correct syntax, to verify if semantic constraints are being satisfied, and to extract relevant information from the input (problem and solutions).

At this point, we need to design some algorithms for solving the problems, i.e. to build LL1, SLR1, LR1 and LALR1 parsers for the language of a given grammar. These algorithms will be invoked at the appropriate time by the *Proletool* language processor using the relevant information extracted as input. These parsers obtained can be used to check the accuracy of the solutions given as input, and use this information to guide the student learning.

### 2.4. General Working Model

*Proletool* users (students and teachers) specify problems as well as solutions, using a formal domain-specific language designed with this purpose in mind. Solutions may not appear in the specification in cases where the student does not want a problem to be corrected but would like to see a possible solution nonetheless. To have this language processed, a processor has been developed. This processor checks if a given input is syntactically and semantically correct, in accordance with the rules of the input language. At the same time, the processor compiles the information needed to understand the problem and the proposed solution, in cases where a solution has been included. It also invokes the algorithms that generate the correct solution to the problem and subsequently checks the level of accuracy of the solution proposed by the student, in cases where a solution has been provided. This process is described by means of the stated diagram shown in Figure 1.

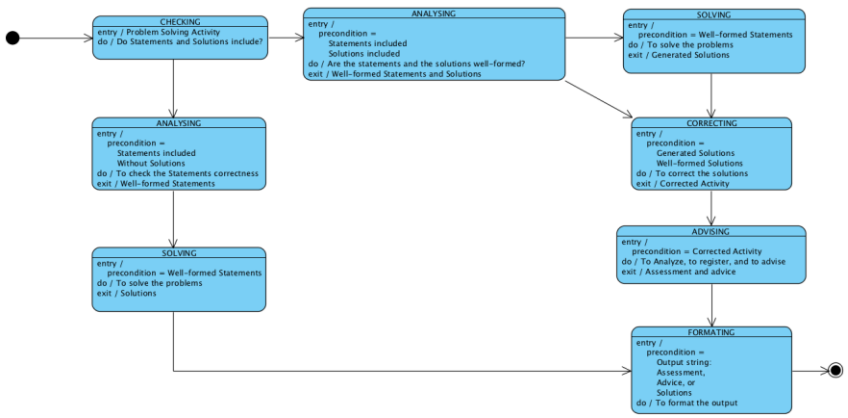



**Fig. 1.** State Machine diagram showing the general working model of *Proletool*

Figure 2 shows a general deployment view of the tool, the core of which is a language processor. The general working of the tool can also be seen. We will use this

diagram to describe how the tool can be used. Also, we will use subsequent diagrammatic figures of the final user interface in order to demonstrate working with the tool. The first step in using *Proletool* is to access it through a web client whereby problems are sent to the tool to be solved (Figure 3).

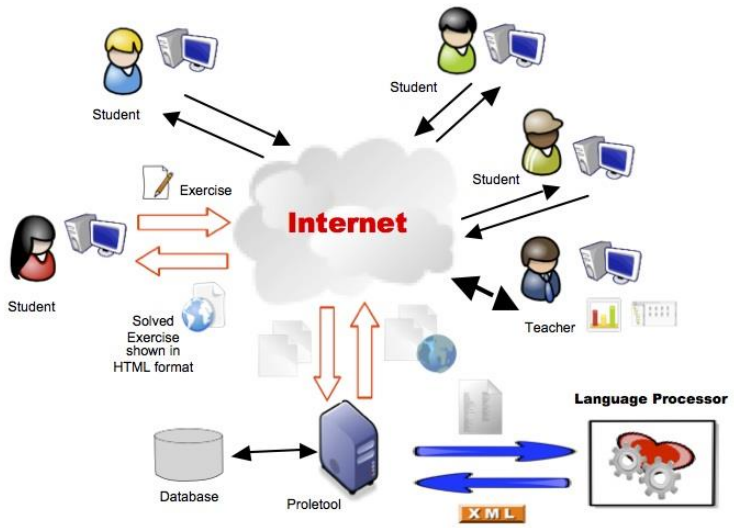



**Fig. 2.** General architecture of the Proletool application

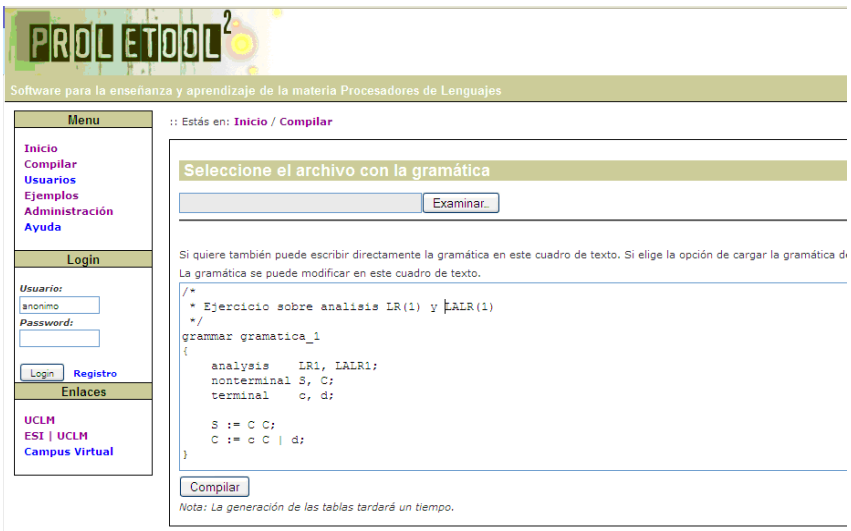
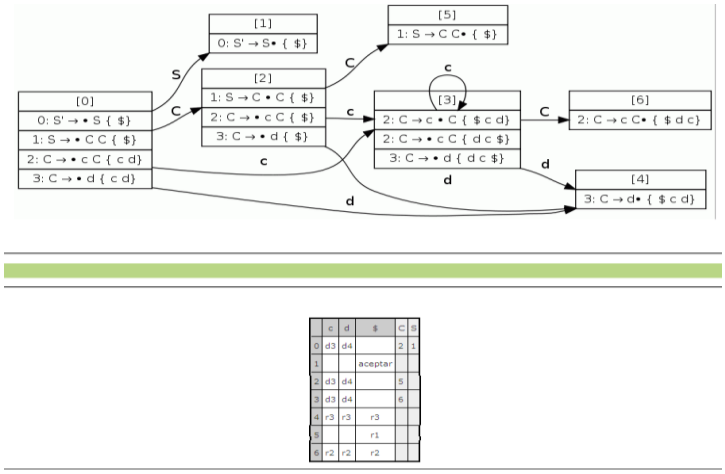



**Fig. 3.** Interface for the specification of problems and solutions

To make the tool more user-friendly for students and teachers, the input of

problems can be done in two ways: a) from files previously written by means of a text editor, which allows users to work off-line and use on-line time for solving the problems; and b) typing in a text area in the tool itself, which is specifically used for the specification of problems, making interaction with the tool easier.




| | c | d | $ | C | S |
|---|---|---|---|---|---|
| 0 | d3 | d4 | | 2 | 1 |
| 1 | | | aceptar | | |
| 2 | d3 | d4 | | 5 | |
| 3 | d3 | d4 | | 6 | |
| 4 | r3 | r3 | r3 | | |
| 5 | | | r1 | | |
| 6 | r2 | r2 | r2 | | |

**Fig. 4.** Output generated by the tool.

In the server, a series of scripts are responsible for invoking the language processor which will analyze the input text, check its accuracy, extract the relevant information, assimilate this information, invoke the appropriate algorithms to obtain a solution and generate the solution in a language that can be visualized, navigated through and stored for later study (Figure 4).

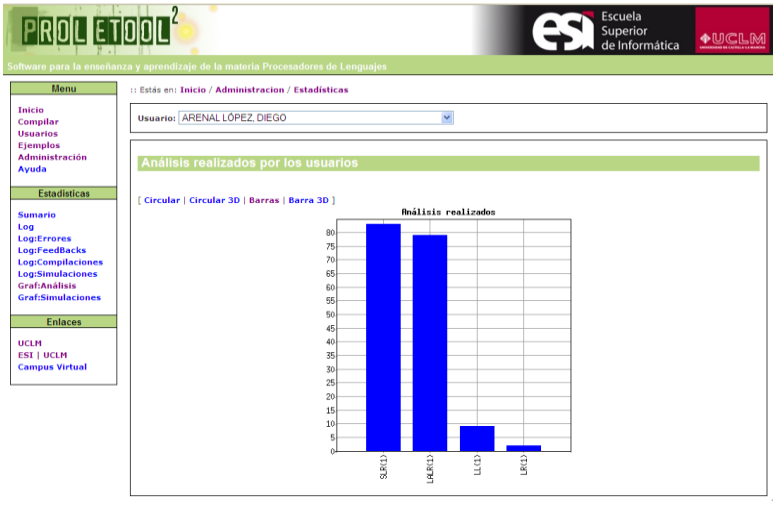



**Fig. 5.** An example of a report generated regarding the activity of a student.

The teacher also has permissions for administration and the possibility of generating reports regarding the activity of students, not only at a general level but also for a specific type of problem (Figure 5). In order to make the tool available for testing and evaluation, and to provide a detailed description of the tool, it can be accessed at the following URL: `http://portal.esi.uclm.es/proletool`.

### 2.5. Language for the Specification of Inputs and Outputs in Proletool

As the communication mechanism suggested to facilitate interaction with the tool is too important, the process followed in developing the *Proletool* domain-specific formal language, that is, the input language for *Proletool*, is described in this sub-section.

As it has already been mentioned, the tool accepts as its input the specification of a set of problems regarding the application of some parsing techniques applied to some grammars, and it subsequently shows the solution to such problems. An input can include one or several problems and, optionally, solutions to those problems can be provided.

A problem consists of a specification of a context-free grammar **G=(V,T,P,S)** where **V** is the set of non-terminal characters, **T** is the set of terminal characters, **P** is the set of productions of the grammar and **S** is the start symbol of the grammar. The grammar is linked to one or several types of parsing (**LL1**, **SLR1**, **LR1** or **LALR1**).

A solution to a problem consists of a table of analysis, which depending on the type of analysis (LL1, SLR1, LR1 or LALR1), will have a different format, together with the information needed to build it.

On the other hand, the output of the tool will be made up by the tables of analysis generated, in addition to:

- Additional information used in the process of solving the problems.
- Important messages obtained during the process.

- Information about conflicts that may happen when the problem cannot be solved.
- The correction of the solution given by the student, if it was provided.

The design of the input language has been carefully executed, because poor definition would have a negative impact in the usefulness of the tool. The language that has been designed incorporates the following general characteristics:

- It uses the terms and the level of abstraction that are more suited to the relevant domain of the problem. There is consistency with the usual notation used in the field.
- Its syntax and semantics are well defined.
- It allows for the composition of *inputs* (descriptions of problems and their solutions) that are as clear and legible as possible, so that their development and subsequent maintenance is easier.
- It allows the specification of one or several problems/solutions at the same time. Consequently, a student can include as many problems as he/she wants in a single file which can then be sent as a single request to the server in order to obtain the solutions. This also permits the specification of several related grammars at the same time, allowing that relationship to be reflected in the solutions obtained.
- It allows the different types of analysis supported (LL1, SLR1, LR1, LALR1) be carried out on the definition of the same entity (grammar) without it having to be redefined.
- It permits the establishment of two scopes of definition, a global scope where common declarations used by several entities (grammars) are specified, and a local scope that is applied to a specific grammar. Symbols that can be globally declared are terminals and non-terminals. Furthermore, groups of productions identified by a name can be specified.

The syntax in the EBNF notation of the language for the specification of the input is the following:

```
CUERPO_GLC       ::= BLQ_GLOBAL BLQS_LOCAL {SOL} | BLQS_LOCAL {SOL}
BLQ_GLOBAL       ::= global '{' CUERPO_GLOBAL '}'
CUERPO_GLOBAL    ::= [DEC_ANALISIS] LISTA_DEC_SIMB LISTA_SENT_GLOB
DEC_ANALISIS     ::= analysis TIPO_ANALISIS [';' TIPO_ANALISIS] [';'TIPO_ANALISIS]
                      [';'TIPO_ANALISIS] ';'
TIPO_ANALISIS    ::= LL1 | SLR1 | LR1 | LALR1
LISTA_DEC_SIMB   ::= {DEC_SIMBOLOS}
DEC_SIMBOLOS     ::= DEC_TERMINAL | DEC_NO_TERMINAL
DEC_TERMINAL     ::= terminal id {',' id} ';'
DEC_NO_TERMINAL  ::= nonterminal id {',' id} ';'
LISTA_SENT_GLOB  ::= {SENT_GRUPO}
SENT_GRUPO       ::= id '=' '{' LISTA_SENT_PROD '}'
LISTA_SENT_PROD  ::= PRODUCCION {PRODUCCION}
PRODUCCION       ::= id ':=' (LISTA_PART_DCHA | ';')
LISTA_PART_DECHA ::= PARTE_DERECHA {'|' PARTE_DERECHA} (';' '|' ';')
PARTE_DERECHA    ::= id {id}
BLQS_LOCAL       ::= DEC_LOCAL {DEC_LOCAL}
DEC_LOCAL        ::= grammar id '{' CUERPO_LOCAL '}'
CUERPO_LOCAL     ::= [DEC_ANALISIS] LISTA_DEC_SIMB LISTA_SENT_LOC
LISTA_SENT_LOC   ::= SENT_LOCAL {SENT_LOCAL}
```

```
SENT_LOCAL       ::= LISTA_SENT_PROD | USE_GRUPO
USE_GRUPO        ::= id ';'
SOL              ::= solution ID '{' PARTES '}'
PARTES           ::= [ANU][INI][SEG][SP]{TABLALL1|TABLA}
ANU              ::= nullable '{' id {',' id} '}'
INI              ::= first '{' DEF_INI_SEG {DEF_INI_SEG} '}'
SEG              ::= follow '{' DEF_INI_SEG {DEF_INI_SEG} '}'
SP               ::= lookahead '{' PROD_SP {PROD_SP} '}'
TABLALL1         ::= ll1_parsing_table '{' CASILLA {CASILLA} '}'
TABLA            ::= TIPO '{' ACTION_TABLE GOTO_TABLE '}'
TIPO             ::= lr1_parsing_table | slr1_parsing_table | lalr1_parsing_table
ACTION_TABLE     ::= action_table '{'{'[' STATE ',' id ']' '=' STATEACTION ';'}'}'
GOTO_TABLE       ::= goto_table '{'  { '[' STATE ',' id ']' '=' STATE ';' } '}'
STATEACTION      ::= 'r' NUMBER | 'd' STATE | 'aceptar'
STATE            ::= NUMBER
DEF_INI_SEG      ::= id '=' '{' id { ',' id} '}' [','']$'] ';'
PROD_SP          ::= NUMBER '=' '{' id { ',' id} [','']$'] '}' ';'
CASILLA          ::= '[' id ',' id ']' '='
                        ( NUMBER ';' | '{' NUMBER {',' NUMBER} '}' ';')
```

The ID symbol matches the *identifier* token, whose construction, given by means of a regular expression, is `ID=[A-Za-z][_A-Za-z0-9]*`

In the right-hand sides of the productions, the declaration of special characters (;,+,*,/,-,...) is also allowed within inverted commas (see the *Proletool* help for a more detailed enumeration). The start symbol of each grammar will be the first non terminal symbol in the first valid production (after expanding the sets of productions inserted into the grammar).

Furthermore, the language we have designed includes a first optional section, which will establish the global scope, and a second section that refers to local scopes, where the definitions of each grammar will appear:

```
//- global section, comment
global {}
//- end of global section
//- local section
grammar problem1 { }
grammar problem2 { }
Sol ejercicio1 {}
//- end of local section
```

Next, we show an example that uses the aforementioned language. This example includes just one problem (with the `Calculator_Problem` identifier), in which one grammar is defined and for which a solution for a LL1 syntactic analysis is requested. Later, it can be seen how the tool provides a solution for this problem. That is, indicating the information needed to obtain the table (`nullable, first, follow`) and the LL1 table itself (`ll1_parsing_table`).

```
grammar Calculator_Problem{
   analysis    LL1;
   nonterminal E, T, F;
   terminal id, num;
     E := E '+' T | E '-' T | T;
     T := T '*' F | T '/' F | F |;
     F := '(' E ')';
     F := id | num ;
}
sol Calculator_Problem{
     nullable{ E, F; }
```

```
first{
        E = {id, num, '(' };
        F = {id, num, '(', ')'};
        T = {id}; }
follow{
        E = {a};
        F = {g};}
ll1_parsing_table{
        [S, a] = 1;
        [F, '('] = 8;
        [F, ')'] = 9;
        [E, '/'] = 1; } }
```

Some semantic considerations should be taken into account with respect to the example:

- Before any symbol is used in a production, it must be declared. However, the special symbols in inverted commas are exceptions to this rule. These symbols are considered as terminals. For example, in the `E := E '+' T` production, the `'+'` symbol is treated as a terminal, but it is not necessary to declare it.
- The type of the symbol is stated by the local declaration, if it exists. If there is no local declaration, the global declaration is considered.
- It is forbidden to re-declare a symbol in the same scope.
- Productions must be the ones of a context-free grammar.
- The use of an identifier for a set of productions locally inside a grammar requires that the identifier has been previously defined in the global area.

The format chosen as the output language of our application is the XML format, as it is a language that is easy to generate and handle by external applications. By using style sheets, XML is integrated in the user interface of the tool (Figure 4), although it could also be used by visual simulators that allow the student to see how the constructed tables can be used (Figure 6).

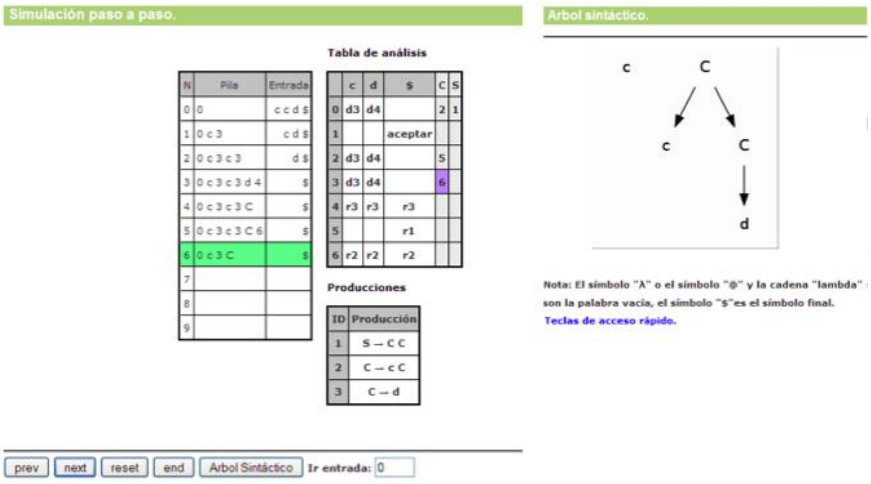



**Fig. 6.** Possibility of graphical interaction with the output obtained.

## 2.6. Results of the use of Proletool

*Proletool* is running since the 2004/05 course and, therefore, we have a lot of registered data. In Table 1, a summary with the most remarkable data regarding the use of the tool since 2004 to the latest academic year (2010/2011) can be seen.

| Variable | 04/05 course | 05/06 course | 06/07 course | 07/08 course | 08/09 course | 09/10 course | 10/11 course |
|---|---|---|---|---|---|---|---|
| **Registered students** | 40 | 77 | 57 | 59 | 29 | 47 | 50 |
| **Server overloads** | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **Processed files** | 678 | 3717 | 2142 | 3262 | 1916 | 1082 | 1126 |
| **Compiled grammars** | 718 | 3884 | 2242 | 3384 | 1942 | 1086 | 1147 |
| **Analysis carried out (total)** | 899 | 5185 | 2670 | 4027 | 2264 | 1249 | 1626 |
| **LL1 analysis** | 282(31,4%) | 1751(33,8%) | 1037(38,8%) | 1971(48,9%) | 1057(46,7%) | 469(37,6%) | 564(34,7%) |
| **SLR1 analysis** | 253(28,1%) | 1466(28,3%) | 557(20,9%) | 707(17,6%) | 339(15,0%) | 322(25,8%) | 468(28,8%) |
| **LR1 analysis** | 302(33.6%) | 1349(26%) | 724(27,1%) | 757(18,8%) | 465(20,5%) | 180(14,4%) | 250(15,4%) |
| **LALR1 analysis** | 62(6.9%) | 619(11,9%) | 352(13,2%) | 592(14,7%) | 403(17,8%) | 278(22,3%) | 344(21,2%) |
| **Simulations carried out (total)** | 74 | 440 | 686 | 560 | 789 | 677 | 253 |
| **LL1 simulations** | 16(21,6%) | 188(42,7%) | 358(52%) | 317(56,6%) | 325(41,2%) | 501(74,0%) | 129(51,0%) |
| **SLR1 simulations** | 9(12,2%) | 127(28,9%) | 62(9,0%) | 79(14,1%) | 62(7,9%) | 29(4,3%) | 36(14,2%) |
| **LR1 simulations** | 38(51,4%) | 68(15,5%) | 216(31,5%) | 68(12,1%) | 70(8,9%) | 10(1,5%) | 61(24,1%) |
| **LALR1 simulations** | 11(14,9%) | 57%(13,0%) | 50(7,3%) | 96(17,1%) | 332(42,1%) | 137(20,2%) | 27(10,7%) |
| **Received comments** | 33 | 42 | 51 | 56 | 21 | 16 | 34 |
| **Average usefulness (1-5)** | 4,2 | 4,3 | 4,4 | 4,6 | 4,6 | 4,8 | 4,8 |
| **Average ease of use (1-5)** | 3,9 | 4,0 | 4,0 | 4,6 | 4,5 | 4,6 | 4,4 |

**Table 1.** Statistics of use of *Proletool*

These data show for disciplines related to language processing, not only that users have used and currently use the tool, but also that they consider it to be useful (see Table 1, the *Average usefulness* variable, which is measured in a range from 1: Useless to 5: Essential) and easy to use (see Table 1, the *Average ease of use* variable, which is measured in a range from 1: Extremely Difficult to 5: Extremely Easy). This has allowed us to check the validity of the approach in the subject Language

Processors, as it seems that it is really possible to make easier the possibility of getting students to state new problems in a systematic way, and also, due the fact that the computer system that supports their resolution is able to give information about the solutions provided. Thus, students can check as many alternatives and variants as they want and, consequently, learning activities in which work is done in the application, synthesis and evaluation levels in the Bloom's Taxonomy are promoted. That is, they can carry out activities that are situated in the higher cognitive levels of this taxonomy and promoting their skills for synthesis, evaluation, etc.

However, the effectiveness of the suggested idea cannot be proven in other "non-linguistic" domains. The main criticism of the proposal is related to the use of a DSL: "The need to learn and use a DSL could represent a hindrance to students or, at least, makes the tool less pedagogically efficiency (as students need to invert time in learning this unknown specific language, which is only useful to interact with the tool itself)". In answer to this objection, we would point out, first of all, that the language complexity is related to the problem complexity. In this sense, complex problems could not be addressed by the suggested proposal. Moreover, the writeability criteria must be fulfilled when the DSL is designed. It must be a language easy to learn and easy-to-use, thought of as an "useful tool" rather than as a Specialized/Technical Language. This will be a responsibility of the designer. More to the point, as it has been noted earlier, authoring tools could be developed and included in the tool, to help student to create his own inputs.

## 3. LESSONS LEARNED: A PROPOSAL OF SOME GUIDELINES FOR THE DEVELOPMENT OF VIRTUAL TEACHING LABORATORIES

In this section we introduce our approach of a generic method or guidelines for designing and developing e-learning applications that provide students with mechanisms to specify their own problems, to add alternative solutions to those problems, and to receive feedback from the system so that they are guided in their learning process. This is the generic method that has been followed in the development of the *Proletool* and *Selfa* (available at http://portal.esi.uclm.es/selfa/) tools, which have been developed to demonstrate and evaluate the method's usefulness. Thus, this method is the main lesson learned we have obtained.

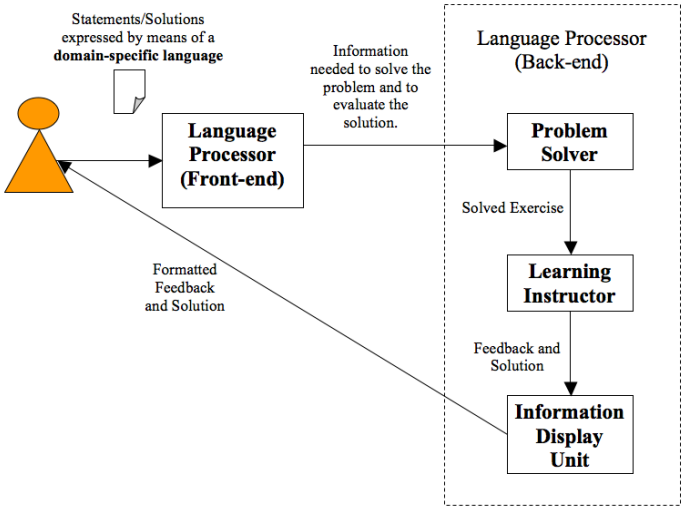



**Fig. 7.** Diagram of the proposed method.

Taking into account the aforementioned key issues, the application of the method can be summarized in the following elements (Figure 7):

- **Formal Language**: This is needed to specify the problem for which the student wants to get a correct solution. Furthermore, it should allow the student to specify the solution that he/she proposes. Obviously, this language is domain-dependent. Formally, it is a Domain-Specific Language and should be designed after a study of the domain has been carried out. The design of a DSL is not a simple task and sometimes the effort may not be worth it.
- **Language Processor**: This is used to process the aforementioned formal language, to acquire the information needed to solve the problem and to evaluate the solution proposed by the student. This processor must validate the input of the applications in a lexical context (by specifying a vocabulary), in a syntactic context (by correcting structures) and in a semantic context (by means of meaningful inputs). Furthermore, it must compile the input information needed to obtain and compare solutions.
- **Problem Solver**: This is the unit used to solve the problems of the domain in which the application works, as well as to explain the solution. For the latter, it must provide an output with all the information needed for the student to understand and analyze the solution generated by the system. The Problem Solver will use the information compiled by the Language Processor to call those algorithms capable of obtaining the solutions to the problems given as an input, and also to check the accuracy of the solutions given by the student in comparison with those obtained by the system.
- **Learning Instructor**: This is used for providing feedback to the student, highlighting what he/she has learned or where a mistake has been made. This

process is based on the result of the comparison carried out by the Problem Solver. The information obtained is used to guide the student in his/her learning process. This unit must have some learning models or paths that establish which steps the students should follow in their learning.

- **Information display unit**: This will be the unit in charge of creating the most suitable display depending on the output generated by system. As one of the purposes of the method is that the student experiments with the output, the display unit could generate animations or provide the student with simulation instruments, thereby enabling concepts to be understood in a faster and more efficient manner. This unit will work with the information provided by the Instructor, which in turn, received the information from the Problem Solver.

In the specific case of the *Language Processor*, that is, the core of the system, we propose an architecture that is shown in Figure 8. This figure shows a diagram of communication including the basic classes as well as the main communication messages. These classes are organized in three packages that correspond to the *Front End*, the *Intermediate* and the *Back End*. Depending on the final implementation, this package organization may be slightly changed.

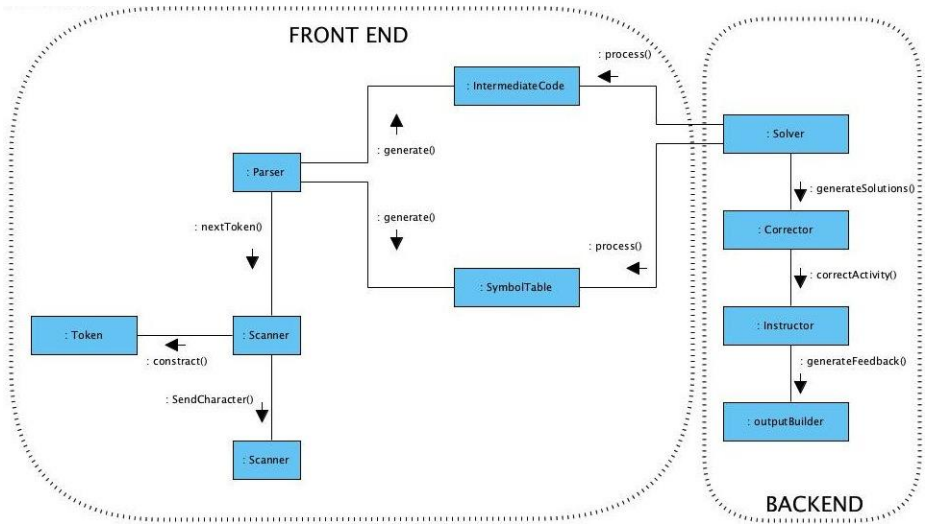



**Fig. 8.** Communication Diagram showing the architecture of the *Language Processor*

Below, some guidelines for the design and construction of learning tools based on the use of formal languages are introduced. Also, we will demonstrate how these guidelines were applied to the design of the *Electronics* learning tool, which is applied in a very different domain. However, due to space limitations, we will only comment briefly the result of each step, whereas a detailed description can be found on the website http://www.esi.uclm.es/jjcastro/electronics. The *Electronics* tool aims to teach students how to solve problems in electrical circuits by means of mesh analysis, using the Kirchhoff's circuit laws.

Below, the steps in the development of *Electronics* are summarized (some

snapshots of the most relevant elements described has been included on the *Electronics* website):

1. Study of the viability of designing a solution that uses a DSL (Mernik, Heering and Sloane, 2005). Two factors must be taken into account when considering the option of developing an application bases on the ideas proposed in this paper:
   - Can the problems be solved by means of an algorithm? That is to say, it should be established whether there exist or not algorithms that can solve problems given as an input, in a reasonable timeframe. If this is not possible, the aforementioned ideas are not applicable to this specific problem.

     The answer to this question in the case of the *Electronics* tool is affirmative, as it is possible to design and implement algorithms for carrying out mesh analysis.
   - Can the input be formalized? That is, it is necessary to establish whether problems and solutions in the field in question can be expressed in a formal way. If it is not possible, the proposal is not applicable to this specific case.

     In the case of the *Electronics* tool, the answer is also affirmative, as it is possible to formalize the input to the algorithm. The process will include a description of the circuit (nodes, their connections and the meshes) and a definition of the elements in it (voltage sources, resistors, capacitors, etc.).

2. Creation of the DSL. In the development of a DSL, two steps can be identified:
   2.1. Analysis. In this step, the domain has to be identified, and knowledge has to be extracted from the available sources. The output of this step will be a domain-specific terminology together with its semantics in a more or less abstract way.
   2.2. Design. In this step, the DSL is designed. In order to accomplish this, its vocabulary has to be designed (lexical part), the rules that allow construction of valid sentences using elements of the vocabulary must be established (syntactic part) and the meaning of the sentences and the constraints that define their validity must be established also (semantic part). The output of this step should be a formal notation of the language. For instance, the Extended Backus-Naur Form (EBNF) meta-language can be used. Furthermore, a table with the elements of the vocabulary (tokens) has to be obtained, together with a definition of the semantics of the language. When carrying out the design step, some properties that are desirable in the language should be taken into account:
      - Clarity, simplicity and unity of concepts. The language should provide users with a set of clear, simple and unified concepts.
      - Well-defined syntax and semantics. With some clear rules, the nonambiguity and the completeness of the language is guaranteed.
      - Consistency with the usual notations. The language should provide elements that mean what they usually mean in the field in which we are working.
      - Legibility. The language should be easy to read and to understand.
      - Ease of use. The language should be easy to understand and use so that its use is encouraged. This property is closely related with the previous one.

To make this step easier, the design can be based on an existing language in the field.
The aforementioned web site includes the EBNF description of the language designed for the *Electronics* tool. The language is split in two: the first part allows the circuit to be defined and the second allows interaction with it by modifying values of the elements defined in the circuit and by posing new questions. Some examples of files written with both languages are also included in the web site.

3. Design and implementation of a Language Processor for the DSL. The design of the language processor should be done, as is usual, in several steps, making a distinction between the front-end and the back-end, thereby facilitating modularity. In the front-end, the input text is analyzed and its validity (at the lexical, syntactic and semantic levels) is checked. The input information necessary to solve the problems and the solutions provided by the students is also compiled in the front-end. The back-end, on the other hand, invokes the algorithms that solve the problems and checks the solutions. The implementation of the language processor can be achieved by using automatic generation tools such as Lex and Yacc (Levin, Mason and Brown, 1998) or ANTLR[6] (AN other Tool for Language Recognition) (Parr, 2007). These tools may be used to obtain the scanner and the parser with semantic actions of the designed language. Another option is to use tools that facilitate the creation, edition and management of programs written with a DSL (Pfeiffer and Pichler, 2008). Yet another option is the one proposed by Microsoft regarding the development of DSLs, called *DSL Tools* (Cook et al., 2007).
   For the *Electronics* tool, a processor has been designed for the first language (definition of the circuit) and an interpreter for the second language (interaction with the circuit). The implementation of both has been achieved using the pair made up by JFlex and Cup, which are the Java equivalent of the Lex and Yacc tools. The source files that have been used to generate the compiler and the interpreter can be found on the aforementioned web site.

4. Design and implementation of the algorithms. Three sub-steps can be identified at this point:
   4.1. Identification sub-step. In this sub-step, the kinds of problems that the tool will solve must be clearly established.
   4.2. Problem solving sub-step. The method that leads us to the solution of each kind of problem has to be established. This refers to the series of finite steps to be followed, including the order of such steps, in reaching a solution.
   4.3. Implementation sub-step.
   Algorithms designed and implemented in the Java language for the *Electronics* tool are available on the web site.

5. Design of the Learning Instructor. To design the Instructor, they must be clear the

[6] http://www.antlr.org

starting point (i.e., what the student already knows), the desired outcome (i.e., the learning objectives), how the tasks are ordered and related (i.e., the sequencing of the learning activities), and finally, a method of evaluating the changes that take place must be established. In order to implement this unit, a mechanism is needed to guide and to record the progress of the student during the learning process by following a given learning design (IMS Global Learning Consortium, 2001; Koper and Tattersal, 2005). Another option could be to use a solution based on the use of finite state machines to model the sequence of objectives in the learning process. In that case, a state change in the machine would occur when changes in the student's level take place. In the *Electronics* prototype, no learning instructor has been included.

6. Design of the Information Display Unit. The design of this unit is dependent on what information is going to be generated and what its use will be. Subsequently, the most suitable display format should be chosen. Animations may be generated, and the tool can be provided with mechanisms that allow the student to interact with the solutions obtained by means of a simulation.
   In the *Electronics* tool, much attention has been paid to this fact. For example, the circuit is graphically shown to facilitate understanding of the solution.

It is worth noting that this proposal could be implemented in a client/server architecture and integrated with a database that allows user management and the collection of data regarding use of the tool. With the latter function, statistics to analyze work carried out by the students could be generated.

## 4. CONCLUDE REMARKS AND FUTURE WORK

In this paper, we have exposed and analyzed the need for designing tools that make students to work in the higher cognitive levels of the Bloom's Taxonomy. We have highlighted the lack of approaches that work in such levels, that is, allowing students to specify new tasks during the learning process, and also, the lack of approaches that are designed following a schema that may be re-used with independence of the learning domain.

To address these problems, in this paper we have presented our experience and as consequence we have proposed some design guidelines that can be useful in certain learning domains for the design and development of learning tools that allow users to specify their own problems, to insert alternative solutions, and to obtain an evaluation from the tool that may guide them through their learning process. This proposal makes extensive use of methods originating in the field of theory of formal languages and their related processors. Thus, a domain-specific language is defined to specify problems and their solutions, and the processor of this language is used to study the validity of the input and to compile the information needed to call the algorithms that will solve the problems and check the accuracy of the solutions.

With the aim of generating interest in our proposal, we have described in detail the *Proletool* application, which is a tool designed to analyze the validity and feasibility of the proposal. The statistics of use of the tool since the 2004-2005 academic year up until the latest academic year, 2010-2011, point out that students consider the tool to be useful and easy to handle. *Proletool* facilitates, in a systematic way, the possibility

for students to propose new problems and for the system to provide them with information about the solutions they provide. This allows students to check as many alternatives and variants of the problem and the solution as they wish. Definitively, learning activities in which work is done in the higher levels of Bloom's Taxonomy are promoted.

In addition, in order to generalize our approach, we have described how to apply these techniques to develop two tools that works in other two different domains, *Selfa-Pro* and *Electronics*. These tools make use of the ideas proposed in this paper to assist learning in the subjects of Formal Languages and Automata Theory and Electronic Circuits, respectively, in the study of Computer Engineering. *Selfa-Pro* is based on the *Selfa* tool (Castro-Schez et al., 2009), the function of which is to make teaching and learning of the subject easier. However, *Selfa-Pro* includes new concepts compared to the original tool. This has required the modification of the input language and the language processor, and the addition of new algorithms to work with the new concepts that have been incorporated. Additionally, the Information Display Unit has been improved. Authoring tools has been included in *Selfa-Pro* tool, where usability has been also improved. It allows the definition of entities and the inclusion of exercises as templates that the student can edit to modify to suit his problem. Thus, it is not necessary to learn the specific DSL for the domain.

With regard to the *Electronics* tool, it is still in the deployment stage, so it is not currently available for evaluation on the web site. However, it does have the aforementioned characteristics and functionality. This tool uses a language to define electrical circuits (a set of interconnected cables, resistors, capacitors, etc.), and also a language to modify their instantiation (by giving values to certain elements) and to pose questions about them, including the option of providing a possible answer. Moreover, an authoring tool will be included in order to facilitate the use of these languages in a visual way.

When in *Electronics* the students throw a question, it is necessary to call the method of solution and analysis of electrical circuits using mesh analysis. Once the solution has been obtained, it is compared with that provided by the students if, indeed, one has been provided. After the comparison has taken place, the tool indicates whether the solution was correct or not. If the student's solution was not specified, the tool will just show the calculated value and an explanation on how it was obtained. The aim of this tool is to show the generalizability of the suggested idea in other area of knowledge different from language processing. This last domain has no relationship with the formal languages domain but it demonstrates that the approach is applicable. Now we are going to register information about its use. Next, we will do a new statistical study to improve our conclusions. However, some snapshots of the most relevant components of this tool are available on its website as we have pointed out.

## Acknowledgements

This work has been partially supported by the Spanish *Ministerio de Economía y*

*Competitividad* within the tin2011-29542-C02-02 project.

## References

ANDERSON, L. and KRATHWOHL, D. (2001): A Taxonomy for Learning, Teaching, and Assessing: A Revision of Bloom's Taxonomy of Educational Objectives., N.Y.: Longman.

BARROS, L.N., DOS SANTOS MOTA, A.P., DELGADO, K.V. and MATSUMOTO, P.M. (2005): A tool for programming learning with pedagogical patterns. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange (Eclipse'05)*, ACM, 125-129.

BLOOM, B.S. (1965): Taxonomy of Educational Objectives, Handbook I: The Cognitive Domain., in, New York: David McKay Co Inc..

BONWELL, C. and EISON, J. (1991): Active Learning: Creating Excitement in the Classroom. *AEHE-ERIC Higher Education Report No. 1*. Washington: Jossey-Bass.

BRAVO, C., VAN JOOLINGEN, W.R. and DE JONG, T. (2009): Using Co-Lab to build System Dynamics models: Students' actions and on-line tutorial advice. *Computers and Education*, 53(2), 243-251.

CASTRO-SCHEZ, J.J., DEL CASTILLO, E., HORTOLANO, J., and RODRIGUEZ, A. (2009): Designing and Using Software Tools for Educational Purposes: FLAT, a Case Study. *IEEE Transactions on Education*, 52(1), 66-74.

CHATLEY, R. and TIMBUL, T. (2005): KenyaEclipse: Learning to program in eclipse. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering (ESEC/FSE'05), 245-248.*

COLE, D., WAINWRIGHT, R. and SCHOENEFELD, D. (1998): Using Java to develop web based tutorials. In *Proceedings of the 29th ACM SIGCSE Tech. Symp. on Computer Science Education*, 92–96.

COOK, S., JONES, G., KENT, S. and WILLS, A.C. (2007): Domain-Specific Development with Visual Studio DSL Tools. Addison-Wesley.

CHURCHES, A. (2008): Bloom's Taxonomy Blooms Digitally. *Tech & Learning*, Accessed 14 November 2011. URL: http://www.techlearning.com/article/8670.

DEWEY, J. (1933): How We Think, A Restatement of the Relation of Reflective Thinking to the Educative Process. D.C. Heath and Company.

EDEN, H., EINSENBERG, M., FISCHER, G. and REPENING, A. (1996): Making learning a part of life. *Communications of the ACM*, 39(4), 40-42.

GORDIJN, J. and NIJHOF, W.J. (2002): Effects of complex feedback on computer-assisted modular instruction. *Computers and Education*, 39(2), 183–200.

IMS Global Learning Consortium (2001). IMS Learning Design, Accessed 19 November 2011. URL: http://www.imsglobal.org/learningdesign/

JOY, M., GRIFFITHS, N. and BOYATT, R. (2005): The boss online submission and assessment system. *ACM Journal on Educational Resources in Computing*, 5(3), 1-28.

JURADO, F., SANTOS, O.C., REDONDO, M.A., BOTICARIO, J.G. and ORTEGA, M. (2008): Providing dynamic instructional adaptation in programming learning.

*Hybrid Artificial Intelligence Systems*, 5271, 329–336.

JURADO, F., MOLINA, A.I., REDONDO, M.A., ORTEGA, M., GIEMZA, A., BOLLEN, L. and HOPPE, H.U. (2009): Learning to Program with COALA, a Distributed Computer Assisted Environment. *Journal of Universal Computer Science*, 15(7), 1472-1485.

KOPER, R. and TATTERSALL, C. (2005): Learning Design: A handbook on modelling and delivering networked education and training. Springer.

KUMAR, A. (2004): Using Online Tutors for Learning - What do Students Think?. In Proceedings of IEEE Frontiers in Education Conference, 524-528.

LEVIN, J. R., MASON, T. and BROWN, D. (1998): Lex & Yacc. O'Reilly, 2nd Edition.

MAKKONNEN, P. (2000): Do WWW-based presentations support better (Constructiristics) learning in the basics of informatics?. In *Proceedings of 33rd Hawaii Int. Conf. on System Sciences (HICSS 2000)*, 1057–1065.

MCCONNELL, J.J. (1996): Active learning and its use in computer science. *ACM SIGCUE Outlook*, 24(1-3), 52-54.

MERNIK, M., HEERING, J. and SLOANE, A.M. (2005): When and how to develop domain-specific languages. *ACM Computing Surveys*, 37(4), 316-344.

OSER, F.K. and BAERISWYL, F.J. (2001): Choreographies of Teaching: Bridging Instruction to Learning. In 'AERA's Handbook of Research on Teaching', 4th Edition, Washington: American Educational Research Association, 1031-1065.

PARR, T. (2007): The Definitive ANTLR Reference: Building Domain Specific Languages. Pragmatic Bookshelf, 1st Ed.

PEREZ, J.R., PAULE, M.P and CUEVA, J.M. (2006): SICODE: A collaborative tool for learning of software development. In *Proceedings of IV International Conference on Multimedia and Information & Communication Technologies in Education* (m-ICTE 2006).

PFEIFFER, M. and PICHLER, J. (2008): A comparison of Tool for Textual Domain-Specific Languages. In *Proceedings of the 8th OOPSLA Workshop on Domain-Specific Modeling*, 1-7.

RODGER, S.H. and FINLEY, T.W. (2011): Jflap project home page. Accessed 14 November 2011. URL: www.jflap.org

SANDERS, D. and HARTMAN, J. (1987): Assessing the quality of programs: a topic for the CS2 course. In *Proceedings of the eighteenth ACM SIGCSE Technical Symposium on Computer Science Education*, 92-96.

SIERRA, J.L., FERNANDEZ-PAMPILLÓN, A.M. and FERNANDEZ-VALMAYOR, A. (2008): An environtment for supporting active learning in courses on language processing. In *Proceedings of 13th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITiCSE'08), 128-132.*

TAMAGNINI, J.J., CAVADINI, S.V., BERDAGUER, P.L., CHEDA, D.A., PACHADO, F. and PETERSEN, M. (2011): Sepa! project home page. Accessed 14 November 2011. URL: http://www.ucse.edu.ar/fma/sepa/

TRUONG, N., ROE, P. and BANCROFT, P. (2005): Automated Feedback for "Fill in the Gap" Programming Exercises. In *Proceedings of the 7th Australasian conference on Computing Education* (ACE '05), Vol. 42, 117-126.