Choosing Grammars to Support Language Processing Courses

Maria João Varanda Pereira[1], Nuno Oliveira[2], Daniela da Cruz[2], and Pedro Rangel Henriques[2]

1 **Polytechnic Institute of Bragança**
Bragança, Portugal
mjp@ipb.pt
2 **Universidade do Minho**
Braga, Portugal
{danieladacruz,nunooliveira,prh}@di.uminho.pt

Abstract

Teaching Language Processing courses is a hard task. The level of abstraction inherent to some of the basic concepts in the area and the technical skills required to implement efficient processors are responsible for the number of students that do not learn the subject and do not succeed to finish the course.

In this paper we intend to list the main concepts involved in Language Processing subject, and identify the skills required to learn them. In this context, it is feasible to identify the difficulties that lead students to fail. This enables us to suggest some pragmatic ways to overcome those troubles. We will focus on the grammars suitable to motivate students and help them to learn easily the basic concepts. After identifying the characteristics of such grammars, some examples are presented to make concrete and clear our proposal. The contribution of this paper is the systematic way we approach the process of teaching Language Processing courses towards a successful learning activity.

1998 ACM Subject Classification K.3.2 Computer and Information Sciences Education, D.3.1 Programming Languages - Formal Definitions and Theory

Keywords and phrases Teaching Language Processing, Domain Specific Languages, Grammars

Digital Object Identifier 10.4230/OASIcs.SLATE.2013.155

1 Introduction

Learning was, is and will be difficult. The student has to interpret and understand the information he got, and then he has to assimilate the new information merging it with his previous knowledge to generate new knowledge.

However teaching is becoming more and more difficult as new student generations are no more prepared to absorb information during traditional classes.

Both statements are true in general, but they are particulary significant in domains that require a high capability for abstraction and for methodological analysis and synthesis. This is the case of Computer Science (CS), in general, and of Language Processing (LP) in particular.

As we will show in the next subsection, many other authors, researching and teaching in LP domain, have recognized the difficulties faced by both students and teachers. To overcome these difficulties, that frequently lead to the nonsuccess and nonsatisfaction of all the participants in the learning activity, and keeping in mind that higher education

2nd Symposium on Languages, Applications and Technologies (SLATE'13).
Editors: José Paulo Leal, Ricardo Rocha, Alberto Simões; pp. 155–168

OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

should focus on improving students' problem solving and communication skills, three main approaches can be identified:

- exploring different teaching methodologies;
- choosing motivating and adequate languages to illustrate concepts and to create project proposals;
- resorting to specific tools tailored to support the development of grammars and language processors in classroom context.

In this paper, we are interested in the second approach. Considering that *a person just learns when involved in a process*, we argue that motivation is a crucial factor to engage students in the course work allowing them to achieve the required knowledge acquisition. In this context, we intend to show that motivation is highly dependent on the languages used to work on during the course. We will discuss the characteristics that a language should have to be a motivating case study. We think that LP teachers should be very careful in their choices and be astute in the way they explore the underlying grammars along the course evolution.

1.1 Related Work

The next paragraphs describe contributions that intend to tackle the problem resorting to different teaching methodologies and techniques.

Li, in [9], states that most topics in a compiler course are quite theoretical and the algorithms covered are more complex than those in other courses. Usually the course content contribute to the lack of students motivation, giving rise to the students unsuccess and to the teacher frustration. The author also thinks that to improve teaching and learning, there are some effective approaches such as concept mapping, problem solving, problem-based learning, case studies, workshop tutorials and eLearning. In particular Problem-based Learning enables students to establish a relation between abstract knowledge and real problems in their learning. It can increase their interest in the course, their motivation to learn science, make them more active in learning, improve their problem solving skills and lifelong learning skills. The problem-based learning is a student-centered teaching approach; it was shown that the approach gets better results when enrolling students that are not at the first year.

Several authors advocate the use of Project-based Learning approaches to teach compilers. Although similar, Project-based and Problem-based Learning are distinct approaches. In Problem-based, the teacher prepares and proposes specific problems (usually focussed in a specific course topic, and smaller in size and complexity than a project) and the students work on each one, over a given period of time, to find solutions to the problems; after that, the teacher provides feedback to the students. In Project-based Learning the students, more than solve a specific problem, have to control completely the project; usually the project covers more than one topic and run over a larger period of time.

Islam et al, in [8], also agree with the complexity of the compiler course and consequently with the students difficulties in this subject. They propose an approach based on templates. Since the automatic construction of compilers is a systematic process, the main idea is to give students templates to produce compilers. The students just have to fill the parts necessary to implement the syntax and the semantics of the language.

Some other authors deal with the problem choosing carefully the language they use for the illustration of concepts or for exercises/projects, as we describe bellow.

Henry has published a paper [7] about the use of Domain Specific Languages for teaching compilers. He says that building a compiler for a domain specific language can engage students more than traditional compiler course projects. In this paper we defend a similar idea. In the cited paper, Henry proposes the use of a new programming language GPL

(Game Programming Language). GPL and the tools provided can be used to create exercises or projects that keep the students motivated because they can define, compile and test video games.

Years ago (1996), Aiken introduced in [2] the Cool Project that was based in an academic programming language used to teach compiler construction topics. Cool (Classroom Object-Oriented Language) is the name for both a small programming language and its processor. Two years later, a language called Jason (Just Another Simple Original Notion) was created by Siegfried [11]. It is a small language based in ALGOL that is used just for academic purposes. Although small, it contains all the important concepts of procedural programming languages that allow the students to extrapolate how to design larger-scale compilers.

Adams and Trefftz propose, in [1], the use of XML to teach compiler principles. They argue that XML processing or Programming Language processing are quite similar tasks, and that a compiler course can be a good place in a Computer Science curriculum to introduce at the same time the main concepts associated to both domains. According to that proposal, the students develop their own grammar and test their project using the tool XMLlint. The authors also describe their experience following that approach.

At last, the next paragraphs refer works that envisage to handle the problem resorting to adequate supporting tools. For that purpose, some compiler construction tools were developed to be used in classrooms.

One of the most significative examples is the work of Mernik et al [10] on LISA system. Using LISA it is possible to use a friendly interface to process Attribute Grammars and generate Compilers (lexical, syntactic and semantic components can be exercised solely or in a whole); useful visualizations are available for each compiler development/execution phase. These visualizations are the key point of LISA; they help students to understand easily the process or the internal structures involved in each phase.

Other examples can be seen in [5]. Demaille et al, introduce in this paper a complete compiler project based on Andrew Apple's Tiger language and on his famous book *Modern Compiler Implementation* [3, 4]. They augmented Tiger language and chose C++ as the implementation language. Considering a compiler as a long pipe composed of several modules, the project is divided in several steps, and students are requested to implement one or two modules. In particular the authors have invested efforts in tools to help students develop and improve their compiler and make the maintenance easier to teachers.

Barrtrada et al [6] combine theoretical and practical topics of the course using diverse modern technologies such as mobile learning, web-based learning as well as adaptive or intelligent learning. They develop a software tool that allows to create learning material for the compiler course to be executed in different learning environments.

The rest of the paper is organized as follows. Section 2 presents the topics that should be taught in an introductory Language Processing course building the correspondent Concept Map, and identifies the requirements that a student must satisfy for achieving the course goals. Section 3 discusses the main difficulties faced by students when attending a LP course. Section 4 introduces our proposal to overcome the difficulties, and defines the characteristics of a language to be considered adequate to support the course being two fold, motivating and enabling to progress incrementally the teaching activity. Section 5 illustrates our proposal, introducing a few examples. Any of those examples are suitable to explain both syntactic or semantic concepts; all of them can be used to support the course evolution, i.e. the introduction of new and more complex concepts, in an incremental mode. The purpose of this section is just to reinforce the approach and offer different alternatives. Section 6 closes the paper with a synthesis of our contribution.

2 Building a LP Course

In this section we define the subjects that should be taught in a *introductory*, *one semester*, **Language Processing course** (also called many times, a Compiler course) that is supposed to appear in the second or third year of an university degree on Computer Science or Software Engineering.

Before identifying the concepts that should be introduced and understood by the apprentices, it is mandatory to define the **learning objectives**.

Learning Objectives

At the end of the course unit the student is expected to be able to work with techniques and tools for formal specification of programming languages and automatic construction of language processors.

More than that, the student should understand the language processing tasks—the main approaches and strategies available for language analysis and translation—as well as the associated algorithms and data structures.

Course Contents

Now we can list the main topics that must be included in the contents of any LP course:

- Programming Language: concept, formal definition, syntax versus semantics, general purpose (GPL) versus domain specific (DSL) languages; examples; Language Design.
- Formal specification of Languages using Regular Expressions (RE) and Grammars (Gr): basic concepts like symbols or tokens of an alphabet, derivation rule or production, derivation tree, abstract syntax tree, contextual condition, attribute evaluation, etc...
- Language Processor: objectives, requirements and tasks; automatic generation tools, like Compiler Generators.
- Lexical Analysis using Regular Expressions and Reactive Automata (coping with symbol names and values).
- Syntactic Analysis using Context-Free Grammars (CFG) and Parsers:
  - Top-Down Parsing, TD (Recursive-Descendant, and LL(1));
  - Bottom-Up Parsing, BU (LR(0), LR(1), SLR(1), LALR(1)).
- Semantic analysis using Translation Grammars (TG) and Syntax Directed Translation (SDT): evaluating and sharing symbol-values, static semantic validation, and code generation using hash-tables and other global variables.
- Semantic analysis using Attribute Grammars (AG) and Semantic Directed Translation (SemDT): attribute evaluation, static semantic validation, and code generation using Abstract Syntax Trees and Tree Traversals.

Notice that part of these topics—those concerned with languages, grammars, and processing approaches or strategies—is more theoretical and will be introduced resorting to formal definitions and algorithms, while the other part—concerned with the implementation of language processors and their automatic generation— is more practical and can be supported by the development of exercises and projects, either manually from the scratch or recursing to tools.

Examples of problems that can be the subject of the above mentioned projects are: text filters; compiler for small or medium size programming languages; or translators for domain specific languages.

2.1 Topics to Learn in a LP Course: a Concept Map

To formalize the knowledge that a student is suppose to acquire in order to achieve the course objectives, we intend to build a Concept Map, or an ontology, describing the Language Processing domain.

The main concepts that we can infer from the course contents presented above are:

- PL – Programming Language;
- GPL – General Purpose Language; DSL – Domain Specific Languages;
- RE – Regular Expression;
- Gr – Grammar; Terminal and Non-Terminal Symbols, Start-symbol, Productions;
- CFG – Context Free Grammar; TG – Translation Grammar; AG – Attribute Grammar;
- LA – Lexical Analysis;
- SynA – Syntactic Analysis (or Parsing)
- TD – Top-Down Parsing; BU – Bottom-Up Parsing;
- SemA – Semantic Analysis;
- CG – Code Generation;
- SDT – Syntax Directed Translation; SemDT – Semantic Directed Translation;
- LP – Language Processor;
- LPG – Language Processor Generator; CG – Compiler Generator
- Interpreter; Analyzer; Compiler; Translator.

Figure 1 is a simplified version of the Concept Map that relates the concepts above in order to describe the knowledge domain under consideration.

2.2 Student skills required to learn LP

From the Concept Map introduced in the previous subsection, we can list the minimum programming skills that a student should have to understand the basic notions and learn the topics involved in a Language Processing course. They are

- knowledge about the basics of computer programming, at least in a imperative (procedural) programming language;
- knowledge about the basic iterative and recursive algorithms;
- knowledge about standard data structures (properties and operations) like list, sets, trees, graphs, tables (matrix) and hash-tables;
- basic knowledge about operating systems and computer architecture.

3 Difficulties faced by Students

When we deal with first year students attending introductory programming courses we know that we need several months to teach a programming language like C, C++ or Java. This happens because students have usually difficulties to interpret the problem statement, to analyze it, to translate what they want to do into an algorithm or a sequence of basic commands or operations. Besides the high level of abstraction required by those tasks, another difficulty arise from the fact that there are several ways to describe the same task in an algorithmic or programming language and the beginner needs to choose the more convenient one. Moreover, to code an algorithm, the student must pay careful attention to all lexical, syntactic and semantic details of the programming as, for instance, the use of semicolons at the end of each statement. There are an high amount of functions and methods spread along a big set libraries or classes that they have to use in an appropriate way. Moreover the students have usually lots of difficulties in algorithm understanding and

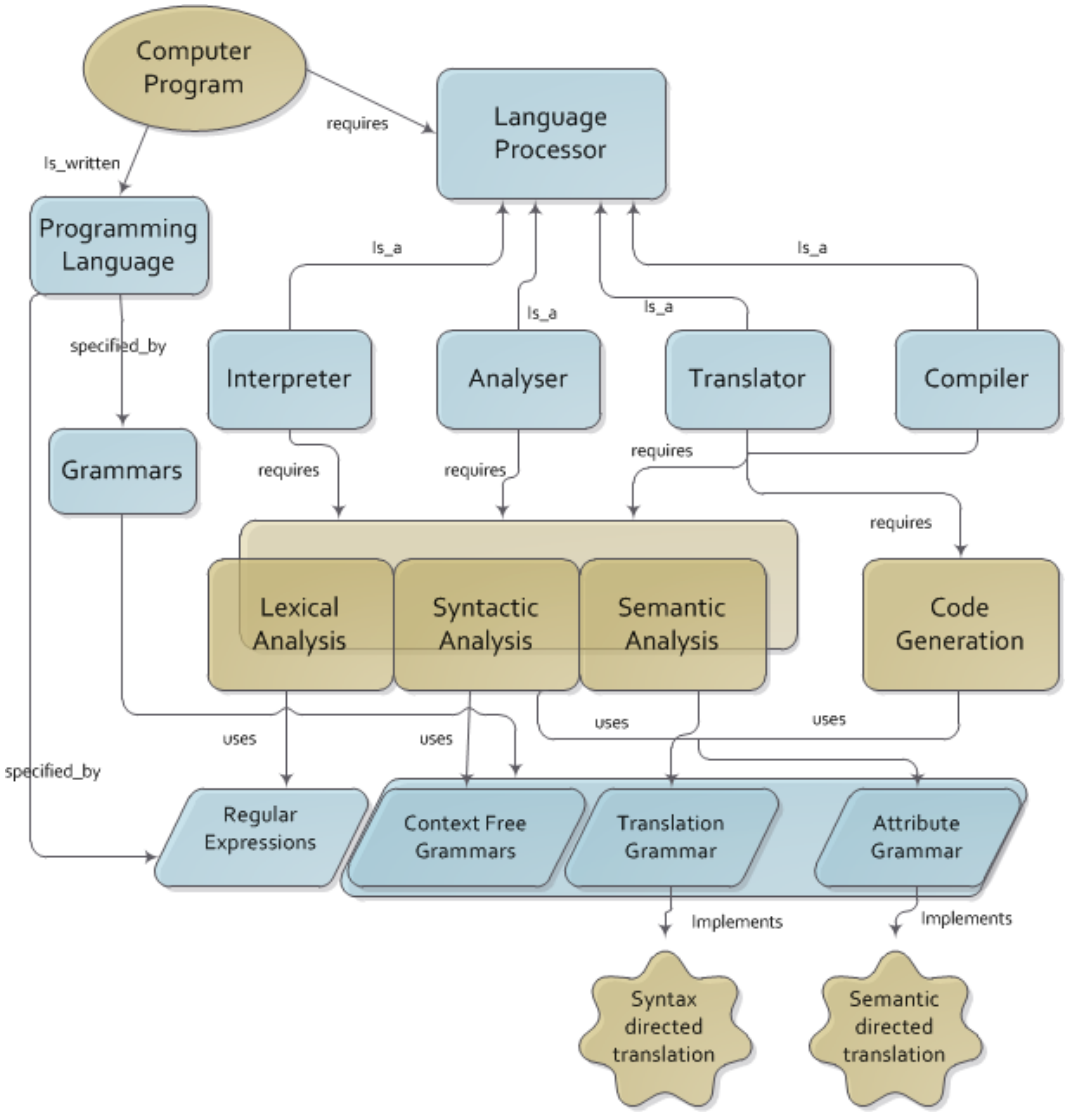

Figure 1 Concept Map describing the Language Processing Domain.

they can not see clearly the relation between the problem and the implementation of the program that is supposed to solve it. There are also data structures that are complex to define and to use.

These are the skills that are at last required for following successfully a Language Processing (LP) course.

In particular, and has remembered before, in LP courses the objective is to teach language principles and compiler construction techniques. For that, we must focus in presenting lexical, syntactic and semantic techniques. These techniques are complex and the students must understand the abstract concepts involved in the problem domain and be able to map them into the program domain concepts.

For instance, the students have difficulties in defining regular expressions since they have a strong expressive power using short specifications. Also the next steps are not easy. Parsing algorithm, attribute evaluation, bottom-up and top-down processes are subjects difficult to teach and difficult to understand.

There are lots of students that when are faced with such difficulties give up. The lack of motivation, due to the field of application traditionally not interesting for most of the students, is responsible for them not go deeply on studding and discontinue the course work.

4 Overcoming the Difficulties: Languages to Support Learning

We have identified the main topics and related concepts that must be taught in a Language Processing course (LPc), and the competences or abilities required to assure students success in such a course. We also identified the common struggles faced by LP learners. In this section we introduce our proposal to overcome the negative factors that lead apprentices to fail.

We assume that the permanent search for new pedagogical methods and techniques, that can be used alone or combined with traditional approaches, is a duty of every teacher in the context of any course. *Problem-based learning* or *Project-based learning* are two examples, discussed in section 1.1, of new methods introduced to improve the students' engagement. Also the resort to eLearning instruments, like forums or collaborative work platforms, is another example of that principle.

We also recognize the relevant role of didactic tools to support LPc. *Grammar Editors*, *Compiler Generators*, *Visualizers and Animators* that allow to follow the generation or compilation processes, are important examples of tools that shall be adopted to ease the students task and help them in understanding the basic concepts.

However our goal is to devise a strategy *to improve students' motivation* as the safest way to get them involved in the course activities helping them to learn with success LP concepts, methods, techniques and tools. With that in mind, we advocate the use of specially tailored languages that will be employed: (i) to illustrate concepts introduced in theoretical classes; (ii) to create exercises to solve in practical classes; and (iii) to elaborate project proposals for students homework.

Based on many years of teaching experience, we believe that this is the most effective approach to overcome the mentioned difficulties, ending up with a high ratio *students-approved/attendants*.

On one hand, we argue that those languages shall be *small* and *simple*. Small is measured in terms of the underlying grammar; a language is said small if the number of non-terminal and terminal symbols is small, as well as the number of grammar productions (or derivation rules). Simple is a twofold characteristic: the objects described by the language shall not be

sophisticated and must be familiar for most of the students; and the tasks involved in the required processing shall be natural and not too complex for understanding or implementing. More than that, we believe that those languages shall possess an incremental character. This is, it shall be possible and straightforward to extend gradually the core language (the language initially proposed) in order to cover more objects in the language domain, or to add requirements concerning the processor output.

On the other hand, we argue that the chosen support languages shall be defined over special domains, instead of being programming languages. These domains must be instinctive for the apprentices, this is, well defined and closed to their common knowledge. In such context, the programs that students are supposed to develop, instead of being traditional *compilers*, will be *translators*—that, for a given input text, produce an output text in a different language—or *generic processors*—that extract data from the source text and compute information to be outputed.

Summing up, we propose the choice of appealing, small and simple, Domain Specific Languages (DSLs), by opposition to the recourse of General Purpose programming Languages (GPLs).

The approach here defended consists in choosing one friendly domain and a simple processing task and then write the grammar for the intended DSL and develop the respective processor. This step will cover the basic lexical, syntactic and semantic concepts. To teach more complex concepts or methods, or to discuss alternative strategies and techniques, the grammar shall evolve covering more domain components or performing more processing tasks. After this stage, other similar and equivalent DSLs shall be used to reinforce all the ideas so far presented.

Concerning project proposals, it is crucial that the language domain is attractive for the students and the project statement is opened enough to give room for their creativity, regarding both the language definition and the processing requirements.

5 Illustrating the Proposal: Examples

In this section we present some language examples to instantiate the approach proposed in the previous section. The examples introduce similar languages than can be used as alternatives to teach grammars (definition and variants, lexical and syntactic issues, static and dynamic semantic aspects of language processing).

Any of these languages are appropriate for an incremental approach enabling the teacher to start with a short and simple problem statement, asking the students to write the grammar (CFG and RE for terminals) and build manually some derivation trees. Then he can elaborate the statement covering more concepts in problem domain in order to extend the grammar. After dealing with the basic lexical and syntactic topics, the teacher can enrich the problem statement adding now some requirements for the desired output leading to the introduction of semantic actions writing the correspondent translation grammar (or, if it is the course objective, to the introduction of attributes, evaluation and translation rules and the correspondent attribute grammar). The requirements can be successively incremented with semantic constraints to introduce validation in semantic actions and error handling (or to introduce contextual conditions in attribute grammars).

All these steps can, and shall, be complemented with practical exercises supported by generating tools.

Each example presented in this section represents a different knowledge domain but in each domain it is possible to create a complete set of exercises tunning the language concerning the desired task.

5.1 1st Example: Lavanda

Lets, then, introduce a domain to work with (and within). Informally, lets think of a big launderette company that has several distributed facilities (collecting points) and a central building where the launder is made. The workflow on this company is as follows: each collecting point is responsible of receiving laundry bags from several clients, and send them to the central building, in a daily basis. The bags are dispatched to the central building with a *ordering note* that identifies the collecting point and describes the content of each bag.

Going deeply, each bag is identified by a unique identification number, and the name of the client owning it. The content of each bag is separated in one or more items. Each item is a quantified set of laundry of the same type, that is, with the same basic characteristics, for an easier distribution at washing time. The collecting points workers should always typify the laundry according to a class, a kind of tinge and a raw-material. The class is either *body cloth* or *household linen*; the tinge is either *white* or *colored* and finally, the raw-material is one of *cotton*, *wool* or *fiber*.

Once in the central building, the ordering notes are processed for several reasons: enter the notes' information into a database, calculate the number of bags received, produce statistics about the type of cloth received, define the value that each client must pay and so on.

Doing such processing by hand is risky because humans are easily error-prone. Therefore, an automatic and systematic way of processing the information in the notes is desirable. A reasonable way of achieving this is use the computer to do the job. In this context, the design of a computer language to describe the contents of an ordering note along with its suitable amount of rules (the grammar) that may be *taught* to a computer is the way to go.

Lavanda is the Domain Specific Language defined in the context of the domain described above, whose main application is to describe the ordering notes that the collecting points of the launderette company daily send to the central building.

Writing grammars according to the domain description requires that the domain concepts and the relations between such concepts are well understood. A good starting exercise is to outline an ontology where the relations between the several domain concepts are expressed. Notice that this approach is feasible due to the *domain size* and consequently, this happens because the domain is a specific one.

Once the domain is studied and internalized writing the grammar is much about giving a concrete shape to the relation between the domain concepts. This shape defines the syntax of the language Figure 2 presents the Context Free Grammar that formalizes the syntax of the language Lavanda.

A valid sentence written according to that grammar is presented below.

```
DAY 2013-03-20 CP Lidl
  BAG 1 CLI ClientA:
       (BODY-COLOUR-COTTON 1 , HOME-COLOUR-COTTON 2);
  BAG 2 CLI ClientB:
       (BODY-WHITE-FIBRE 10)
```

The grammar in figure 2 has enough complexity to propose exercises concerned with recursivity (lists), lists of lists, keywords and alternative productions.

```
p1:          Lavanda    →  Header  Bags
p2:          Header     →  DAY date CP IdCP
p3,p4:       Bags       →  Bag  |  Bags ';' Bag
p5:          Bag        →  BAG num  CLI IdCli ':' '(' Items ')'
p6,p7:       Items      →  Item  |  Items ',' Item
p8:          Item       →  Type  Quantity
p9:          Type       →  Class '-' Tinge '-' Material
p10:         IdCP       →  id
p11:         IdCli      →  id
p12:         Quantity   →  num
p13,14:      Class      →  BODY |  HOME
p15,16:      Tinge      →  WHITE    |  COLOUR
p17,18,19:   Material   →  COTTON   |  WOOL    |  FIBER
```

Figure 2 Lavanda Grammar.

Some examples of output requirements that can be formulated in this context are:

- compute the total of items delivered by each client;
- compute the total of bags in the order;
- compute the total of items in the class body clothes and total in the class household line;
- verify that there are not client identifiers occurring more than once.

It is also possible to add more productions to grammar in order to cope with some other concepts like prices, washing times and scheduling. This allows to add more complexity to the exercise and more tasks can be proposed like: compute the amount to be paid by each clients, consult the daily scheduler and the processing state of each bag, generate the invoices for each client, generate html code to construct a web page with the information involved, and so on.

5.2 2nd Example: Genea

This second example shows how in a completely different, but still common sense, domain we can define a language with characteristics similar to the previous one. We believe that students can be engaged in the exercises proposed around this subject.

Lets, then, introduce a second domain to work with (and within). A research organization devoted to demography and history has a complex application that constructs genealogical trees from simple specifications of families and offers a lot of statistics, computations and relation-based information. Family records consist of the basic part of each family which is the parents and their children. As it is obvious, dates play an important role in history, therefore born, death and wedding dates are important in this domain.

All persons are identified with their first name, but only the parents (a father or a mother) have their family names (as before marrying). Children hire their family name from the father. In contrast with the parents, children must have their gender defined.

Although small and very well defined, this domain is full of common-sense restrictions and relations that need to be respected. The most important ones concern chronological order, and age-related issues.

Regarding this simple domain, the researchers decided to build a language capable of specifying each family. The language should be processed by the application to construct the tree and to make information and computations available to the users of the application.

```
p1:     Genea     →  Families
p2,p3:  Families  →  Family  | Families ';' Family
p4:     Family    →  Parents WED Wedding CHILDREN Children
p5:     Parents   →  Parent Parent
p6:     Parent    →  Type ':' Name Name Life
p7,p8:  Children  →  &  |  Children Child
p9:     Child     →  Gender Name Life
p10:    Life      →  '(' Born '-' Death ')'
p11,12: Type      →  FATHER | MOTHER
p13,14: Gender    →  MALE | FEMALE
p15:    Born      →  date
p16,17: Death     →  date | '?'
p18:    Wedding   →  date
p19:    Name      →  id
```

Figure 3 Genea Grammar.

Genea was the language defined based on this domain. Its concrete context free grammar is presented in Figure 3. Notice that the empty string symbol is denoted by &.

A sentence of the grammar is expressed below to show a concrete and correct source text.

```
FATHER : Herman Einstein (1847.08.30 - 1902.10.10)
MOTHER : Pauline Koch (1858.02.08 - 1920.02.20)
WED 1876.08.08
  CHILDREN
  MALE Albert (1879.03.14 - 1955.04.18)
  FEMALE Maja (1881.11.18 - 1951.06.25)
FATHER : Albert Einstein (1879.03.14 - 1955.04.18)
MOTHER : Mileva Maric (1875.12.19 - 1948.08.04)
WED  1903.01.06
  CHILDREN
  MALE Hans (1904.5.14 - 1973.07.26)
  MALE Eduard (1910.07.28 - 1965.10.25)
```

The grammar in figure 3 uses recursivity to represent lists of lists but the inner lists can be empty. Some interesting outputs can be produced from these lists. This language also uses some keywords, special characteres and non−literal terminals like date that would allow to propose some exercises related with their intrinsic value.

Some examples of output requirements that can be formulated in this context are:

- compute the number of children in each family, total and separated by gender;
- compute the total of families in the description;
- compute the average age at death;
- compute the mother's age at the first birth (average);
- generate an SQL statement to insert each child in a database; the child's family name is obtained concatenating the mother's surname with the father's surname;
- generate dot specifications in order to visualize the family tree;
- verify that the death date is greater than the birth date;
- verify that the wedding date lies within the birth and death interval.

5.3 3rd Example: Orienteering Paths Planner

Foot Orienteering is a widely developed sport in Portugal. Basically, an athlete receives a map with a marked path; in that path there are signaled control points that must be visited in the required order; at the end of the course, the athletes return to the start point and are scored according to control points visited and also according with the time spent. In each contest, competitors are divided by age class. A different path is given to each age class.

In order to help the organization of competition, we propose a new DSL to specify the list of paths (each path will be, therefore, a list of control points), so that the distance can be calculated and the course be visualized.

The required language should start by identifying all the control points of a given area where the competition takes place. Each point will be identified with an acronym and its Cartesian coordinates. Also, the language should enable us to define each path, indicating its name, age class, and list of points (described by acronyms). The order in the list establish the visiting order.

OPPL was the language defined based on this domain. Its concrete context free grammar is presented in Figure 4.

A sentence of the grammar is expressed below to show a concrete and correct source text.

```
POINTS
  A(3,5)
  B(4,2)
  C(5,5)
  D(9,9)
  E(5,15)
PATHS
  soft (>10) (A,B,C)
  medium (>20) (A,C,B,D)
  hard (>20) (A,E,C,D,B)
```

The grammar in figure 4 uses recursivity to represent two indepent lists. This allows to propose different exercises for each list. There are also some keywords and special characteres as in the other examples.

Some examples of output requirements that can be formulated in this context are:

- compute the total number of points and/or paths;
- compute the number of points in each path;
- compute the distance between two points;
- compute the length of a path;
- generate dot code to visualize the paths.

```
p1:     OPPL    →  POINTS Points PATHS Paths
p2,p3:  Points   →  Point  | Points Point
p4:     Point    →  letter '(' num ',' num ')'
p5,p6:  Paths    →  Path | Paths Path
p7:     Path     →  name Age '(' List ')'
p8:     Age      →  '(' '>' num ')'
p9,p10: List     →  List ',' letter | letter
```

Figure 4 OPPL Grammar.

It is also possible to add more productions to the grammar in order to cope with the athlete information. In this context new symbols must be created representing names, numbers, paths, time spent and scores of each athlete. More exercises can be proposed with this new information; new output results can be required, like athletes ranking, partial scores, historical results and so on.

6 Conclusion

The use of DSLs in teaching methodologies allows to chose a knowledge domain appropriated to the students. When students are aware of the domain, its main concepts and relations, it is much easier to explain and discuss the processing of a language in that domain. In this sense, the efforts made to explain a subject like language processing do not dependent any more on the complexity of GPL grammars.

The usual grammar size of DSLs is more appropriate for teaching when compared with GPLs. Smaller grammars allows the students to understand better the concepts involved. Moreover, these kind of languages can be easily changed, adapted or incremented depending on the complexity of the example that the teacher desires to show and discuss with students.

Working within these small and common sense domains we can hope that the students quickly and easily guess the processing results expected for given source text samples. This allows to check if the language processing is well done or if there something that must be tuned.

Our proposal differs from the others in the sense that we do not create a special language to support our teaching activities. Instead we present the characteristics that a language, and its grammar, should exhibit to be helpful. Besides that, we systematize how to take profit of the toy languages chosen to introduce different topics and evolve from a concept to the next concept, in a smooth and challenging way in order to keep students interested and engaged.

We did not perform an evaluation experiment but we have been using this approach during the last 10 or 15 years and the results are truly positive. We achieved an average of 80% of students approved over students assessed. The grades reached by the students in the practical works prove that they are motivated, they adquired the basic language processing concepts, they were able to apply them and they had an opportunity to use their criativity.

References

1 D. Robert Adams and Christian Trefftz. Using XML in a compiler course. *ACM Sigcse Bulletin*, 36:4–6, 2004.

2 Alexander Aiken. Cool: A portable project for teaching compiler construction. *Sigplan*, 1996.

3 Andrew W. Appel and Maia Ginsburg. *Modern Compiler Implementation in C*. Cambridge University Press, 2004.

4 Andrew W. Appel and Jens Palsberg. *Modern Compiler Implementation in Java*. Cambridge University Press, 2002.

5 Akim Demaille, Roland Levillain, and Beroit Perrot. A set of tools to teach compiler construction. *ACM SIGCSE*, 40(3):68–72, 2008.

6 M.L. Barron Estrada, Ramon Zatarain Cabada, Rosalio Zatarain Cabada, and Carlos A. Reyes Garcia. A hybrid learning compiler course. *Lecture Notes in Computer Science*, 6248:229–238, 2010.

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

8 Md. Zahurul Islam and Mumit Khan. Teaching compiler development to undergraduates using a template based approach. *Bangladesh*.

9 ZhaoHui Li. Exploring effective approaches in teaching principles of compiler. *The China Papers*, 2006.

10 Marjan Mernik and V. Zumer. An educational tool for teaching compiler construction. *IEEE Transactions on Education*, 46(1):61–68, 2003.

11 Robert M. Siegfried. The jason programming language, an aid in teaching compiler construction. *ESCCC*, 1998.