Table of Contents

Pedro Rangel Henriques
University of Minho, Portugal

Chapter 8

A Sports Science Approach to Computer Programming Education

Costa Neves, M.
Ramires, M.
SportTools - Technology for Sport Company, Portugal
Carvalho, J.
School of Education of the Polytechnic Institute of Setúbal, Portugal
Piteira, M., Santos
J., Folgôa
N., Boavida, M.
School of Technology of the Polytechnic Institute of Setúbal, PortugalMaria João Varanda Pereira

Section 3: Frameworks and Tools

Chapter 11

ZatLab: Programming a Framework for Gesture Recognition and Performance Interaction

André Baltazar

Luís Gustavo Martins

UCP - School of Arts, Center for Science and Technology in the Arts - Porto, Portugal

Chapter 12

Design a Computer Programming Learning Environment for Massive Open Online Courses

Ricardo Queirós

Center for Research in Advanced Computing Systems (CRACS/INESC-TEC);

AN EFFECTIVE WAY TO TEACH LANGUAGE PROCESSING COURSES

An Effective Way to Teach Language Processing Courses

Maria João Varanda Pereira[1], Nuno Oliveira[2],
Daniela da Cruz[3], Pedro Rangel Henriques[3]

[1] Instituto Politécnico de Bragança, CCTC – UM
[2] Universidade do Minho, HASLab/INESC-TEC
[3] Universidade do Minho, CCTC/DI - UM

Abstract

In this chapter we identify the difficulties that lead students of Language Processing (LP) courses to fail. All of us that teach Language Processing topics are aware of the complexity of this task; we know that a great part of the students will face big difficulties inherent to the level of abstraction associated with some of the basic concepts in the area, and to the technical capacities required to implement efficient processors. A key issue that we have identified along the years we are teaching Language Processing courses is the lack of students' motivation to learn the main topics.

This issue is something that we want to overcome with our proposal.

A starting point for this research is to identify the main concepts involved in Language Processing subject, and to understand the skills required to learn them. Considering that *a person just learns when is 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

Table of Contents

Pedro Rangel Henriques
University of Minho, Portugal

Chapter 8

A Sports Science Approach to Computer Programming Education

Costa Neves, M.
Ramires, M.
SportTools - Technology for Sport Company, Portugal
Carvalho, J.
School of Education of the Polytechnic Institute of Setúbal, Portugal
Piteira, M., Santos
J., Folgôa
N., Boavida, M.
School of Technology of the Polytechnic Institute of Setúbal, PortugalMaria João Varanda Pereira

Section 3: Frameworks and Tools

Chapter 9

Ensemble - an innovative approach to practice computer programming

Ricardo Queirós

José Paulo Leal

Center for Research in Advanced Computing Systems (CRACS/INESC-TEC);

Department of Computer Science, Faculty of Science, University of Porto, Portugal

Chapter 10

Moodle-based tool to improve teaching and learning of relational Databases design and SQL DML queries

M. Antón-RodríguezM.A. Pérez-

JuárezM. I. Jiménez-Gómez

F.J.

Díaz-PernasM.

Martínez-Zarzuela

D. González-Ortega

Department of Signal Theory and Communications, and Telematics Engineering,

Telecommunication Engineering Technical School, University of Valladolid, Spain

Chapter 11

ZatLab: Programming a Framework for Gesture Recognition and Performance Interaction

André Baltazar

Luís Gustavo Martins

UCP - School of Arts, Center for Science and Technology in the Arts - Porto, Portugal

Chapter 12

Design a Computer Programming Learning Environment for Massive Open Online Courses

Ricardo Queirós

Center for Research in Advanced Computing Systems (CRACS/INESC-TEC);

knowledge acquisition. We will state that motivation is highly dependent on the languages used to work on during the course. So, we 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.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 particularly 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 sequel, many other authors, researching and teaching in LP domain, have recognized the difficulties faced by both students and teachers. To overcome these difficulties, which frequently lead to the failure and dissatisfaction of all the participants in the learning activity, and keeping in mind that higher education 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.

As previously introduced in [17], our focus is the second approach. Considering that *a person just learns when he is 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 chapter, we 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. LP teachers should choose carefully the sample languages used to explore the underlying grammars along the knowledge transfer process.

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 contributes to the lack of student's motivation, giving rise to the student's fail and to the teacher frustration. 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, and improve their problem solving skills and lifelong learning skills. Problem-based Learning is a student-centered teaching approach; however, it was shown [9] that the approach gets better results when enrolling students that are not at the first year.

Project-based Learning is another relevant approach 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 focused 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 below.

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 chapter we uphold and recommend 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 Notation) 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.

Some other authors, for instance Mernik and Demaille, handle the problem resorting to adequate supporting tools. For that purpose, some compiler construction tools were developed to be used in classrooms.

In this trend, one of the most significant 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.

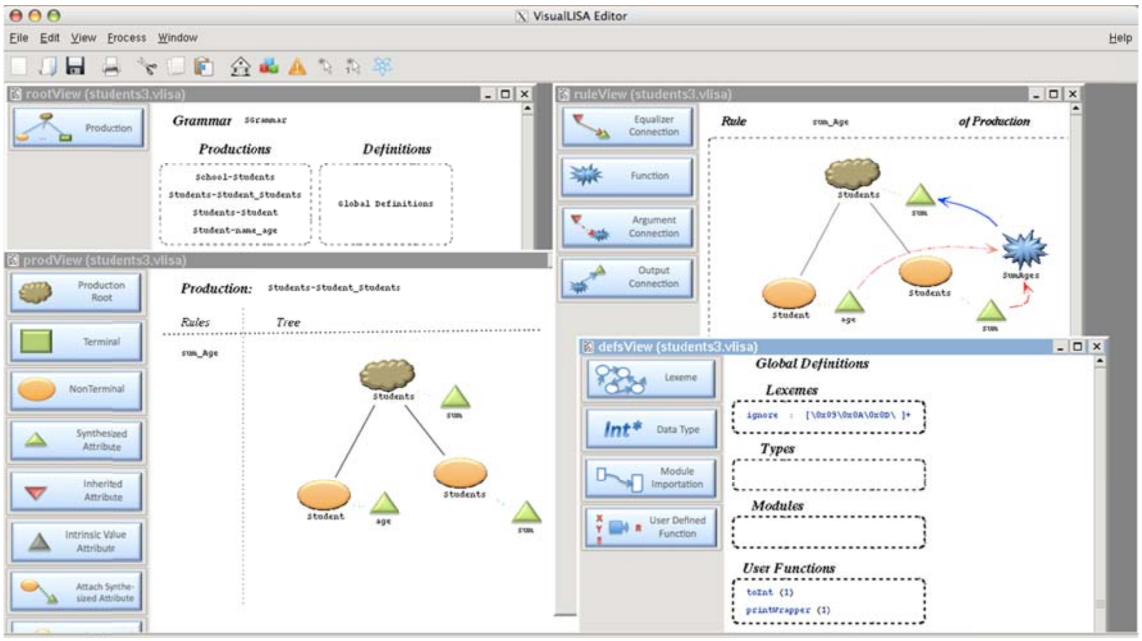

Figure 1. VisualLISA editor

VisualLISA, Oliveira et al [15, 16], is a visual interface for LISA, as depicted in *Figure* ***1***. Users can pick up, from the left-hand side dock (see *Figure* ***1***), grammar icons and build up attributed productions with the associated evaluation rules and contextual conditions. To assure legibility and cope with scalability, the visual environment is completely modular and production oriented---this is, each window corresponds to a production (grammar derivation rule), and more than one production window can be created to allow for the separate definition of different attribute evaluation rules. VisualLISA is strongly recommended to be used in the context of Language Processing Courses due to the natural way an attribute grammar can be *drawn* after it is imagined. After drawing a new grammar, VisualLISA editor generates a readable textual LISA description, for further processing, and also generates an intermediate XML representation (called XAGRA) also very easy to read and understand. For more details, please look at http://www3.di.uminho.pt/~gepl/VisualLISA/.

Other example of this tool-based teaching approach can be seen in [5], where Demaille et al introduce 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.

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.

Our proposal differs from the others in the sense that we do not create a special language to support our teaching activities. Instead 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.

The chapter will be organized as follows. Section 2 presents a Concept Map that describes the main topics that should be taught in an introductory Language Processing course, and identifies the requirements that a student must satisfy for achieving the course goals. As a consequence, Section 3 discusses the difficulties felt by students when attending a LP course. Then Section 4 introduces our proposal to overcome the difficulties, and defines the characteristics of an adequate language that is, on one hand, motivating, and, on the other hand, that enables to progress incrementally the teaching activity. To illustrate our proposal, in Section 5 we introduce five case studies that will allow us to discuss teaching matters from lexical to syntactic and semantic concerns. The paper ends in Section 6 with a synthesis of our contribution.

2. Building a LP Course

In this section we define the subjects that should be taught in an 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 a university degree on Computer Science or Software Engineering.

Before identifying the concepts that should be introduced by the teacher 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:

- Languages and Programming Languages: concept, formal definition, syntax versus semantics, general purpose (GPL) versus domain specific languages (DSL); examples; Language Design.
- Formal specification of Languages using Regular Expressions (RE).

- Formal specification of Languages using Grammars (Gr): symbols or tokens of an alphabet, derivation rule or production, derivation tree, abstract syntax tree, contextual condition, attribute evaluation, etc...
- Language Processors: objectives, requirements and tasks; automatic generation tools (*Lexer*, *Parser* and *Compiler Generators*).
- Lexical Analysis using Regular Expressions and Reactive Automata; dealing with symbols (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.

Part of these topics—those concerned with languages, grammars, and processing approaches—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 resorting 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.

Topics to learn in a LP course: a Concept Map

From the course content, presented above, we can infer the main concepts that characterize that area (knowledge domain):

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))

SemA – Semantic Analysis

CG – Code Generation

SDT – Syntax Directed Translation;

SemDT – Semantic Directed Translation

LP – Language Processor; Interpreter, Analyzer, Compiler, Translator

LPG – Language Processor Generator or CG – Compiler Generator

To formalize that knowledge (whose items were listed above), that a student is supposed to acquire in order to achieve the course objectives, we built a Concept Map, or an ontology, describing the *Language Processing Domain*, as shown in *Figure 2*.

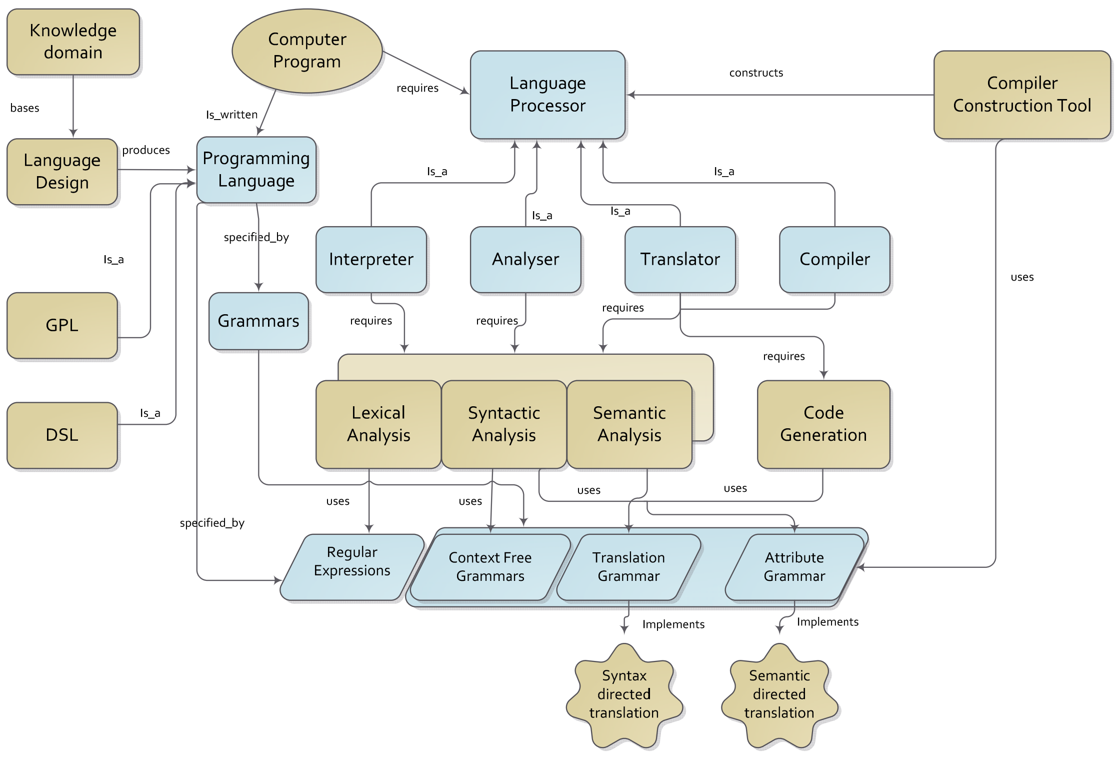

Figure 2. A concept map describing the LP knowledge domain

Student skills required to learn LP

From the Concept Map introduced in the previous subsection, we can identify the minimum programming skills that a student should have to understand the basic definitions 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 *lists*, *sets*, *trees*, *graphs*, *tables (matrix)* and *hash-tables*.

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 language. There are a high amount of functions and methods spread out 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 they cannot 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 least required for following successfully a Language Processing (LP) course.

As remembered before, in LP courses the objective is to teach language/grammar theory and principles as well as 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. In practice we have observed that 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* or *attribute evaluation algorithms, bottom-up* and *top-down processes* are subjects difficult to teach and difficult to understand.

There are lots of students that, when faced with such difficulties, give up. As students are not motivated---due to the fact that the application field is not interesting for most of them---they do not go deeply on studding and discontinue the course work.

4. Overcoming the difficulties: Languages to support learning

We have identified the main 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.

Our group also advocates the use of Automatic Grading Systems (AGS). The authors are, for some years, deeply involved in the development of an AGS (Quimera), as can be seen in [12, 13, 14]. AGS in the context of the teaching/learning process are two fold tools. On one hand, they support teachers assessing in an effective and fair way students belonging to big classes where problem- or project-based learning was adopted; on the other hand, they stimulate students to proceed on, as they provide detailed and immediate feedback (after the completion of a given exercise).

We also recognize the relevant role of didactic tools to support LP teaching/learning as mentioned in the Introduction. *Grammar Editors, Compiler Generators, Visualizers and Animators* (that enable to follow the generation or compilation processes) are important examples of tools that shall be adopted to ease the students' task and to 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 student's 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 high ratio students-approved/attendants. In the last ten or fifteen years, considering medium sized classes with an average number of 150 students, we measured that around 50% of the attendants are assessed (this is, 50% of the class students complete all the assessment duties) and that around 90% of them are approved.

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. Usually domain specific languages use keywords (literal terminals) that are strongly related with domain concepts which makes easier the relation between program and problem domains. 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 outputted.

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 main idea is to start explaining a specific domain and then use a DSL already created or create a new one. The advantage of this approach is based on the fact that everyone knows what kind of things will be expressed by the program written with this DSL and the teacher can concentrate his efforts in explaining how to specify the language using a grammar or how to build a language processor using a compiler construction tool. The students can train the specification of lexical and syntactic parts and they have no problem to understand the semantic rules they have to define because the map between the program and the problem domain is more intuitive. The teachers can tune each DSL example in order to include more or less complexity depending on the course objectives. They can incrementally add features to the new language and show how to overcome each difficulty.

The approach here recommended 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. Then at a first stage, the teacher can ask the students to write the grammar (CFG and RE for terminals) and build by hand 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).

These steps shall be complemented with practical exercises supported by generating tools.

1st Example: Book Index

The first example is concerned with book indexes. The main idea is to define a book title and for each page a set of topics that can be found in that page. So, the concepts involved in this domain are: book, page, title and special term (topic).

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 (a conceptual map) 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 relations among the domain concepts. This shape defines the syntax of the language.

The graph shown in ***Figure* 3** defines the ontology for this case-study.

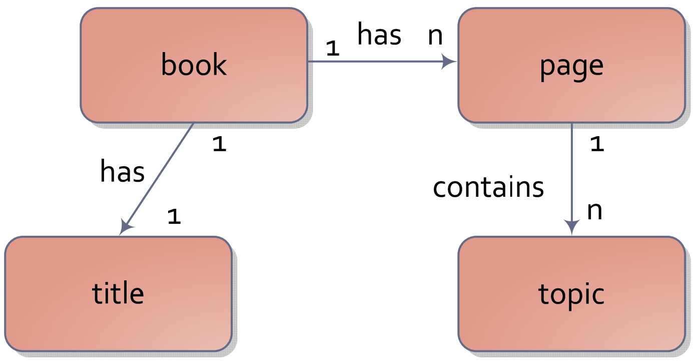

Figure 3. Book index ontology

For each book, a title and one or more pages can be specified; and for each page one or more topics can be associated. The concrete context free grammar is the following:

```
p1:    Book   → Head Lines '.'
p2:    Head   → INDEX Title
p3:    Title → String
p4,p5: Lines  → Line | Lines ';' Line
p5:    Line   → Page '=' Topics
p6:    Page   → num
p7,p8: Topics → Topic |Topics ',' Topic
p9:    Topic → String
```

The grammar allows for specifying the syntax of the language. The source program written in this language is divided in two parts: a header and a body (`Lines`). The header has a reserved word (`INDEX`) and the title of the book. The body is composed of one or more lines and each line defines the set of topics for one page. The page is defined as a number and a topic is defined as a `String`. A sentence of the grammar is expressed below to show a concrete and correct source text:

```
INDEX "Sample Book 1"
1 = t1,t2,t3;
2 = t1,t4,t2,t5;
3 = t4,t3,t2,t6.
```

Although the grammar is very simple this exercise allows for proposing different tasks to the students that are not so simple. Some examples of output requirements that can be formulated in this context are:

- 1st level tasks (use only atomic global variables, no need to store intermediate values, and simple semantic actions):
  - compute the total number of pages
  - compute the total number of different terms (or topics)
- 2nd level tasks (require intermediate and complex data structures and more elaborated semantic actions):
  - verify that there are not repeated pages
- 3rd level tasks (require still more complex data structures to store intermediate values and more sophisticated semantic actions):
  - generate an output with the desired Book Index following the structure below:

```
Index
t1: 1, 2.
t2: 1, 2, 3.
t3: 1, 3.
t4: 2, 3.
```

To perform these tasks the student need to associate appropriated semantic actions to each grammar production in order to update the counters (1st level tasks), to directly produce an output (1st level tasks) or to save the information in an intermediate structure. Working over that structure it is possible to count distinct topics (2nd level tasks) and to produce an output that shows the information in a different order and format (the `Index` produced in the 3rd level task).

2nd Example Shopping List

The second example is also a very simple one that has a short statement and small domain, but that exhibits some complexity typical of *list languages* like the programming language Lisp. It is also a common sense domain that does not require a long explanation but that is prone to involve the students.

Consider that someone wants a very simple language to describe his/her shopping list, which can be composed of one item, or more items. Items have just a product name and they can appear isolated or grouped. Each group represents a sub-list of products that belong to the same category.

This domain can be described by the ontology depicted in *Figure **4***.

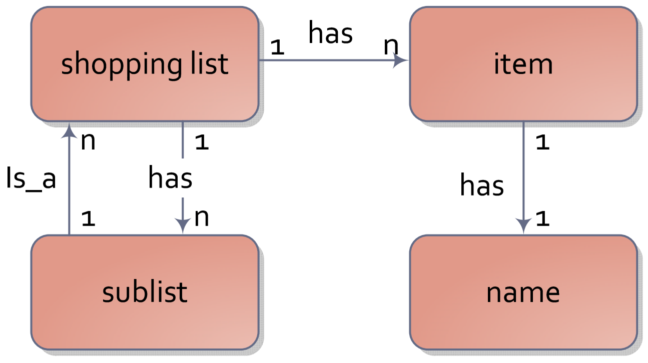

Figure 4. Shopping list onntology

Notice that the idea is to create sub-lists of items inside the global shopping list but each such sub-list follows exactly the same syntax of a shopping list.

The concrete context free grammar that derives from the description above is the following:

```
p1,p2:      ShoppingList → Item | '(' SubList ')'
p3:         SubList      → ShoppingList OtherS
p4,p5:      OtherS       →  | ',' SubList
p6:         Item         → Name
p7:         Name         → String
```

This grammar is strongly recursive and must be carefully explained to the students. One of the best ways to do this is to present a valid sentence of the grammar and ask the students to construct the derivation tree. In this case, it must be clear that the grammar allows to specify lists composed of items or lists separated by ','. Each `item` has a `name` that is a String.

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

```
( rice, (wine, beer, water), paper, (showergel, soap) )
```

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

- 1st level tasks:
  - compute the total number of items
  - compute the maximum list length
- 2nd level tasks:
  - verify that there are no repeated items
- 3rd level tasks:
  - generate a dot file to draw a hierarchical structure of items

In the 1st level tasks, the counters can be computed "on the fly"; it is possible and enough to associate simple semantic actions to productions `p2` and `p6` to directly produce the result. In the 2nd level, an intermediate data structure must be filled during the parsing (by the semantic actions associated to the productions); at the end (after parsing the input file), the items repeated shall be removed from that intermediate structure. The task at the 3rd level uses that intermediate structure to collect the information needed to generate the output file.

Notice that the three difficulty levels correspond, like in the previous example, to the complexity of the data structures and of the semantic actions that are required to produce the desired output. We believe that the successful achievement of the tasks in one level will motivate students to improve their code and proceed to the next phase, learning more about the construction of language processors.

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 classes; a different path is given to each age class (corresponding to different difficulty levels).

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 domain for this problem is described by the ontology depicted in *Figure* **5**.

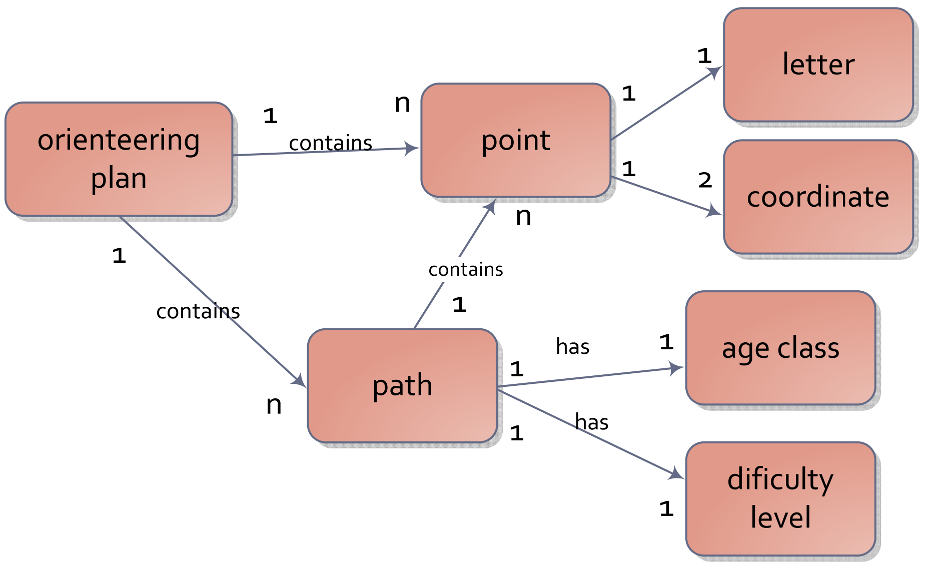

Figure 5. Orienteering paths planner ontology

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 difficulty level (soft, medium or hard), age class, and list of points (described by acronyms). The order in the list establishes the visiting order.

OPPL was the language created based on the domain defined above. Its concrete context free grammar is the following:

```
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        → Level Age ’(’ List ’)’
p8:          Age         → ’(’ ’>’ num ’)’
p9,p10:      List        → List ’,’ letter | letter
p11:         Level       →  SOFT | MEDIUM | HARD
```

This grammar allows for clearly separating the point specifications from the path specifications. A list of (one or more) points and a list of (one or more) paths are the main parts of the grammar. Each point has a letter and a pair of numbers (that define its coordinates). Each path has a level (implemented as an enumerated variable), an age (represented as the *greater than* ‘>’ character and a number) and a list of point identifiers (denoted as letters).

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)
```

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

- 1st level tasks (require simple global variables and simple semantic actions):
  - compute the total number of points
  - compute the total number of paths
  - compute the number of points in each path
- 2nd level tasks (concerning intermediate data structures, nothing special is needed but the semantic actions should be more elaborated):
  - compute the length of a path
- 3rd level tasks (as in the previous examples, these tasks require complex intermediate data structures to store the data collected from the input text in order to build the output code):
  - generate dot code to visualize the paths

This example allows the teacher to convince students how easy it is to solve an apparently complex (but really interesting) problem, when following its structural definition. This is, when adding the semantic actions to the appropriate syntactic rules.

In this case, both 1st and 2nd level tasks can be performed updating the counters and printing the results directly in each production. As happens with the other examples, the 3rd level task needs an intermediate structure; such structure allows for collecting the necessary information during parsing, which is then used to produce the output.

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

4th Example: Lavanda

Let’s, then, introduce a domain to work with (and within). Informally, let’s 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, sending them to the central building in a daily basis.

The bags are dispatched to the central building with an ordering note that identifies the collecting point, the date and describes the content of a set of bags.

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 point’s 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, and 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 to use the computer to do the job. In this context, the design of a computer language to describe the contents of an ordering note supported on a formal grammar is the way to go.

Lavanda is the Domain Specific Language defined in the context of the domain described rigorously by the ontology in *Figure **6***.

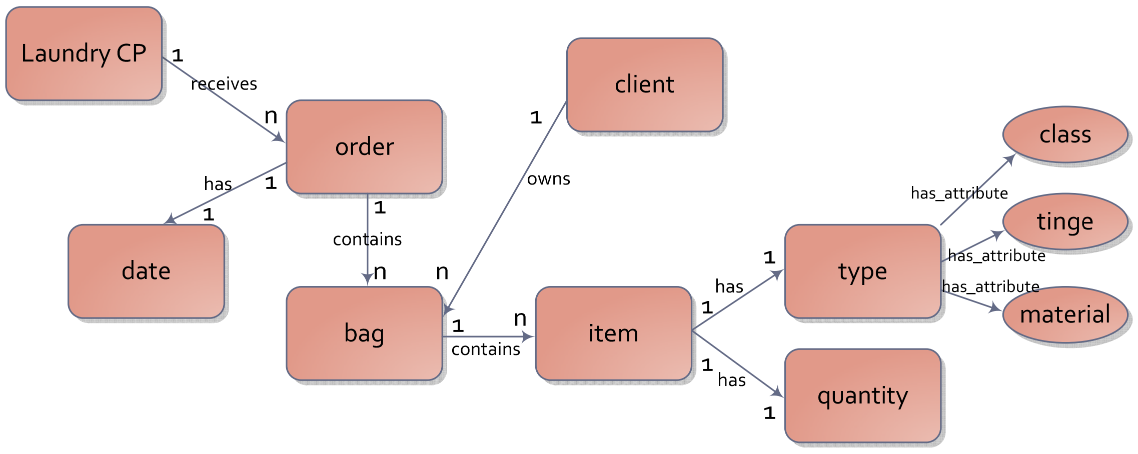

Figure 6. Lavanda ontology

The main purpose for design this language is to develop a tool that automatically creates the ordering notes that the collecting points of the launderette company daily send to the central building.

The Context Free Grammar that formalizes the syntax of the language Lavanda, according to the ontology drawn, is the following:

```
p1:           Order        → 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 | HOUSE
p15,16:       Tinge        → WHITE | COLOR
p17,18,19:    Material     → COTTON | WOOL | FIBER
```

This grammar represents the information concerned with one order. Each order is defined by a header and a set of bags. Each bag has a number, a client identification and a set of items. Each item is defined by a type and a quantity.

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

```
DAY 2013-03-20 CP Lidl
BAG 1 CLI ClientA:
(BODY-COLOR-COTTON 1 , HOUSE-COLOR-COTTON 2)
```

```
BAG 2 CLI ClientB:
(BODY-WHITE-FIBRE 10)
```

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

- 1st level tasks (once again this tasks can be solved using just atomic global variables and simple semantic actions):
  - compute the total of bags in the order
  - compute the total of items in the class body clothes
  - compute the total of items delivered by each client
- 2nd level tasks (the tasks in this group require the use of complex data structures to store intermediate values and more complex semantic actions):
  - find the client with the biggest number of white items
  - verify if there are two bags with the same number
  - order the bags by number

This example is longer than the previous ones and has a statement a bit more complex, but allows the teacher to show how a convenient structural definition enables the development of simple semantic actions to perform sophisticated transformations.

In this case, only two levels of tasks are proposed: a 1st level including tasks that are solved using direct translation and a 2nd level where the results must be computed after parsing. It is also possible to add more productions to the grammar in order to cope with some other concepts like prices, washing times and scheduling. This allows adding more complexity to the exercise and more tasks can be proposed like: compute the amount to be paid by each client, 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.

5th Example: Genea

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

Let's, then, introduce another 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, described by the conceptual map (or ontology) shown in *Figure* **7**, the researchers decided to build a language capable of specifying each family.

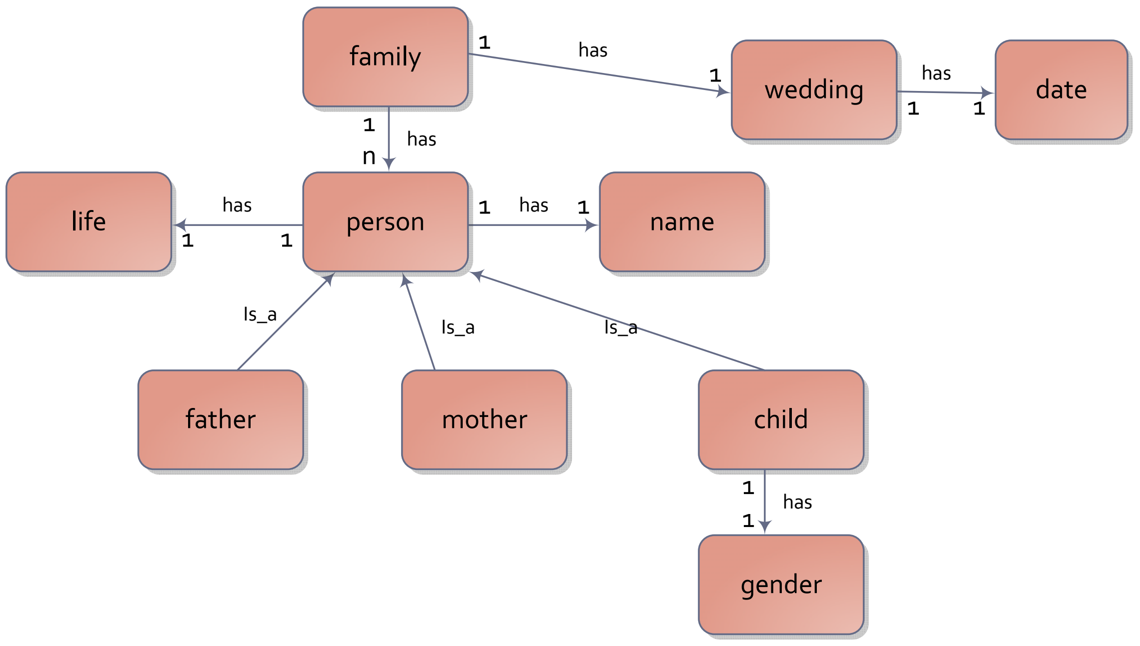

Figure 7. Genea ontology

This conceptual map has a novelty concerned with the use of the relation 'is_a' between two concepts: `father is_a person`, `mother is_a person` and `child is_a person`. This taxonomic relation has some influence when deriving the grammar, as it will be shown below.

Genea was the language defined based on this domain. The concrete context free grammar is the following (notice that the empty string symbol is denoted by &):

```
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
```

This grammar doesn't strictly follow the conceptual map, because its final design depends on the actual context in which the language will be used and depends also on the inspiration of each designer. In the grammar it is possible to define exactly the number of parents (that should be necessarily 2). The restriction that one of the parents should be a `father` and the other one should be a `mother` was not taken into account in the syntactical level; it will be imposed at the semantic level. Moreover, in the conceptual map, `name` and `life` are associated with Person concept but in the grammar they were repeated in two productions, one that is concerned with parents and another that is concerned with child, because the concept `Person` was not considered useful as a symbol of the grammar.

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)
```

In the following list, some examples are presented of requests formulated in the context of this exercise.

- 1st level tasks (use only atomic variables and simple semantic actions):
  - compute the number of children in each family, total and separated by gender
  - compute the total of families in the description
- 2nd level tasks (these tasks can also be solved with atomic variables but require more elaborated semantic actions and the storage of intermediate values):
  - compute the average age at death
  - compute the mother's age at the first birth (average)

    - concatenate the mother's surname with the father's surname so that children hire both family names
- 3rd level tasks (here the difference is mainly conceptual; these tasks allow the introduction of contextual conditions that must be checked to validate the semantics of the source text):
    - Verify for each family that the death date is greater than the birth date
    - verify for each family that the wedding date lies within the birth and death interval with respect to both parents
- 4th level tasks (require more complex intermediate data structures to store the extracted data in order to produce the desired output code):
    - generate dot specifications in order to visualize the family tree
    - generate an SQL statement to insert each child in a database with all the respective info (including child's surname)

Lots of interesting exercises can be proposed in the context of this language. Four sets of tasks are presented here with an incremental level of complexity.
This example is appropriate to motivate students because on one hand it deals with common sense concepts, and on the other hand enables the generation of a real database from a high-level source description, validating the input data. It can also be easily extended with more syntactic rules and more semantic actions.

6. Conclusion

Teaching Language Processing courses (or Compilers) for more than twenty years, the authors have a solid knowledge about the topics that must be introduced, and they are aware of the difficulties such topics rise on the students. They also know well how these difficulties are in general heavy to be overcome due to the pupils' lack of motivation to study grammar theory or compiler classic subjects. Looking every year to improve the academic indicators (like the ratios *approved/assessed* or *assessed/attendants*) of the teaching success associated to those courses, the authors found a way they are systematically following each year. They strongly believe that the approach introduced and discussed along this chapter---the choice and adoption of appropriate Domain Specific Languages to motivate apprentices---is promising.

The use of DSLs in teaching methodologies allows choosing 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 allow the students to understand better the concepts involved. Moreover, Domain Specific 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 the given source text samples. This allows to easily test the language processor developed and decide whether it is well implemented or contains errors that must be fixed.

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 have been applying this approach the last ten years at Universidade do Minho and Politécnico de Bragança. The results obtained are encouraging. The average percentage of successful students (when doing all practical exercises and the theoretical exam) is 85%. This means that students, who decide to attend the classes and do the practical exercises, become motivated and involved enough to also study the underlying theory, attaining the minimum requirements to be approved. Unfortunately if we consider the percentage of students who register in the course and those who actually do all the assessment work demanded, we obtain every year a smaller figure (around 60%). This, however, happens in all the other courses of the same academic year.

References

1. Adams, D. & Trefftz C. (2004). Using XML in a compiler course. *ACM Sigcse Bulletin*, 36, 4–6.
2. Aiken, A. (1996). *Cool: A portable project for teaching compiler construction*, Sigplan.
3. Appel, A. & Ginsburg, M. (2004). *Modern Compiler Implementation in C*, Cambridge University Press.
4. Appel, A. & Palsberg, J. (2002). *Modern Compiler Implementation in Java*, Cambridge University Press.
5. Demaille, A., Levillain R. & Perrot, B. (2008). A set of tools to teach compiler construction, *ACM SIGCSE,* 40(3), 68–72.
6. Estrada, M., Cabada, Ra., Cabada, Ro. & Garcia, C. (2010). A hybrid learning compiler course, *Lecture Notes in Computer Science*, 6248, 229–238.

7. Henry, T. (2005). Teaching compiler construction using a domain specific language, *ACM SIGCSE*, 37(1), 7–11.
8. Islam, Md. & Khan, M. Teaching compiler development to undergraduates using a template based approach, Bangladesh.
9. Li, Z. (2006). Exploring effective approaches in teaching principles of compiler, *The China Papers*.
10. Mernik, M. & Zumer, V. (2003). An educational tool for teaching compiler construction, *IEEE Transactions on Education*, 46(1), 61–68.
11. Siegfried, R. (1998). The jason programming language, an aid in teaching compiler construction, *ESCCC*.
12. Fonte, D., Vilas Boas, I., Cruz, D., Gançarski, A. & Henriques, P. (2012). Program Analysis and Evaluation using Quimera, *ICEIS'2012 --- 14th International Conference on Enterprise Information Systems,* 209-219.
13. Fonte, D., Cruz, D., Gançarski, A. & Henriques, P. (2013). A Flexible Dynamic System for Automatic Grading of Programming Exercises, OASIC.SLATE.2013 --- *Symposium on Languages, Applications and Technologies,* 29, 129-144.
14. Fonte, D., Vilas Boas, I., Oliveira, N., Cruz, D., Gançarski, A. & Henriques, P. (2014). Partial Correctness and Continuous Integration in Computer Supported Education, *CSEdu'2014 --- 6th International Conference on Computer Supported Education,* (to be published).
15. Oliveira, N., Henriques, P., Cruz, D. & Varanda Pereira, M. (2009). VisualLISA: Visual Programming Environment for Attribute Grammars Specification, *Proceedings of the International Multiconference on Computer Science and Information Technology -- 2nd Workshop on Advances in Programming Languages (WAPL'2009),* 689-696.

16. Oliveira, N., Varanda Pereira, M., Henriques, P., Cruz, D. & Cramer, B. (2010). VisualLISA: A Visual Environment to Develop Attribute Grammars, *ComSIS -- Computer Science an Information Systems Journal, Special issue on Advances in Languages, Related Technologies and Applications,* 7 (2), 266-289.

17. Varanda Pereira, M., Oliveira, N., Cruz, D. & Henriques, P. (2013). Choosing Grammars to support Language Processing Course, OASIcs.SLATE.2013 --- *Symposium on Languages, Applications and Technologies,* 29, 155-169.

Terms

CFG (Context Free Grammar) - is a grammar that only defines the syntax of a language. A CFG is a four tuple composed of a set of Terminal symbols (T) that belong to the language Vocabulary, a set of Non-terminal symbols (N) that are abstractions of sub-sentences with special meaning, a set of Derivation Rules or grammar Production (P) that define how each Non-terminal symbols is expanded into other Non-terminal or Terminal Symbols, and the grammar Axiom or Start-symbol (S) that is the Non-terminal from which all valid sentences must derive. This is, a sequence of Terminal symbols in T is a valid sentence of the language defined by a given CFG if and only if that sequence derives from S applying the grammar Derivation Rules in P.

DSL (Domain Specific Language) - is a kind of Formal Language defined to be used in a restrict domain for a special purpose.

GPL (General Purpose Language) - is a kind of Formal Programming Language created to be used to instruct computers to solve problems in general domains (not only in a specific one). For instance, a problem in the area of Physics can be solved using the same GPL as another problem in the area of Natural Language Translation or Accounts.

Grammar - is a formalism to define rigorously the syntactic and the semantic rules of a Language. The Sender that needs to write a sentence in that language must follow the grammar rules to create a valid sentence, and the Receiver must also follow the same grammar to interpret it, this is, to extract the sentence's meaning.

Language - a set of sentences, being a sentence a sequence of symbols that belong to a given alphabet or vocabulary. Symbols are combined according to a set of syntactic and semantic rules. Each sentence has a meaning, and Languages are used to communicate transporting messages from a Sender to a Receiver. Natural Languages (NL) are used when both Sender and Receiver are Human Beings and Programming Languages (PL) are used to control machines, i.e., the Receiver is a machine (for instance, a Computer). PL are Formal Languages because syntactic and semantic rules are defined rigorously prior to their first use.

Language Processing - is the analysis and translation (or transformation) of sentences of a given language. This process is compulsory so that the computer can understand (extract the meaning) the sentences written and provided by the Programmer in order to execute them, this is, to perform the actions described by the processed sentences.

Language Processing Course - is a set of lectures where the students learn theoretical concepts and pratical aspects about programming languages formalization and implementation. The first topic is concerned with the rigorous design and specification of languages, and the second topic is concerned with methods and techniques to interpret (process) those languages using a computer.
Usually these courses also teach how to construct automatically the referred language processors based on the formal specification of the language. The practical work proposed make the students capable of building concrete processors.

Language Processor - is a computer program, a software tool (as, for instance, a Compiler or an Interpreter) aimed at processing the sentences of a given Language. Given a sentence (the so-called input or source-text), a Language Processor must analyze it to check if it is valid according to the grammar rules, and to extract its meaning; after that analysis phase (composed of three layers; lexical analysis, syntactic analysis and semantic analysis), if no errors are found, the Language Processor transforms the input into the desired output (maybe a target-text or a value).

Ontology - An ontology formally represents knowledge as a set of concepts (general concepts or classes, and occurrences of the concepts, or instances) within a domain, and the relationships among those concepts (hierarchical or taxonomic relations, and non-taxonomic relations). It can be used to reason about that domain described through the definition of its objects their properties and relations.

TG (Translation Grammar) - is a CFG extended with Semantic Actions to define also the semantics of the underlying language. Semantic Actions are pieces of code that are attached to each Production in P (the set of the CFG Derivation Rules) to describe the semantic constraints that must be observed by the symbols in a concrete sentence and also to define how the sentences must be transformed or processed.