Applying a Competency- and Problem-Based Approach for Learning Compiler Design

Ahmed Khoumsi and Ruben Gonzalez-Rubio
University of Sherbrooke

Abstract

Our department has redesigned its electrical engineering and computer engineering programs completely by adopting a learning methodology based on competence development, problem solving, and the realization of design projects. In this article, we show how this pedagogical approach has been successfully used for learning compiler design.

KEYWORDS
Competence development, Problem-based Learning, Knowledge acquisition, Assessment, Compilation, Lexical analysis, Syntactic analysis.

1. INTRODUCTION

Several studies have shown serious gaps between the objectives of university engineering programs and the needs of an evolutionary economy [1]. As a solution, our department has undertaken a fundamental and major reform of its Bachelor of Electrical and Computer Engineering degrees [2]. The new learning approach is based on *competence development* for *solving problems* and *realizing design projects.*

After an introduction to the new learning method, we illustrate its application in the computer engineering program, by a problem-based learning (PBL) unit which aims at learning *compiler design*. This PBL unit is denoted APP_{comp}[1]. This paper is aimed essentially at people interested in new learning approaches in computer engineering. Note that the PBL approach described here has also been applied to learn many other subjects. For example, [3] presents a PBL unit for learning probabilities.

This paper is structured as follows. In Section 2, we introduce the pedagogical approach adopted in our department. Sections 3 and 4 present APP_{comp} as an illustration of the PBL approach. In Section 3, we present the competencies, the necessary knowledge for developing such competencies, and the documentation used as resources in APP_{comp}. Section 4 presents the contextual problem to be solved and the various pedagogical activities realized in APP_{comp}. In Section 5, we explain how students are assessed in APP_{comp}. Section 6 discusses the advantages of the PBL approach in learning compiler design. And in Section 7, we conclude the paper.

2. PEDAGOGICAL APPROACH AND ORGANIZATION

In this section, we present the main principles behind the major reform to our electrical and computer engineering programs. A more detailed presentation can be found in [2], from which this section is inspired. Such reform aims at making the objectives of university engineering programs compatible with the needs of economy and society [1].

2.1 Competency and knowledge

"Conventional" engineering programs give priority to knowledge acquisition. With the reform, priority is given to the development of *competencies*. Simply speaking, a competency can be seen as an ability to act and use *resources*, for solving a given task. Note that competency is not synonymous with know-how, because the competency is flexible and adaptable, and cannot be reduced to an algorithm. Competency concerns more heuristics than algorithms. In our reformed engineering programs, competencies are classified in four types: scientific and technical competencies, design competencies, inter-personal competencies, and intra-personal competencies.

Development (or implementation) of a competency requires acquisition of knowledge, which can be considered as resources. Knowledge has been classified into three types: declarative (know factual information), procedural (know *how to use* factual information), and conditional (know *when and where to use* factual information). In the context of our engineering program, factual information can consist of, for example, a definition, a theorem, a hypothesis, a rule, or an algorithm.

2.2 Organization of a semester, PBL approach

The programs are organized around four-month periods which, for simplicity, will be called *semester*[2]. The programs last eight *academic semesters*, alternating with four *industrial training semesters* beginning after the third academic semester. Each semester is based on a theme (e.g., computer systems, embedded systems, etc.) and includes two types of activities: six consecutive two-week problem-based learning (PBL) units, and a design project which is spread over the whole semester (see Fig. 1). With a total of 15 credits per semester, the project is worth 3 credits in each of the first six semesters, and 6 credits in each of

[1] APP is the French acronym for Apprentissage par Problèmes et par Projets.
[2] Henceforth, the term semester means academic semester.

the last two semesters.

Each of the six two-week PBL units of a semester is based on a problem to be solved, rather than on a discipline or subject as in a conventional program. This approach is motivated by the fact that PBL is the natural mode of knowledge acquisition. A problem must come from a real engineering situation, but also be presented in such a way that the students have to identify their existing (i.e., previously acquired) knowledge and the new (i.e., not still acquired) knowledge, that are necessary for solving the problem. The formulation of the problem must also lead the students to identify the necessary skills for solving the problem effectively. This learning *contextualization* provides realistic situations where knowledge is applied, and thus, encourages a better understanding of that knowledge.

PBL encourages *active* learning, and thus, students are more *responsible* and *autonomous* in the learning process. Professors are "resources" that react by providing opinions or indications, validating or invalidating solutions, asking questions, etc. But professors should never provide a solution (or information allowing to deduce straightforwardly a solution).

Let us consider Semester 3 of the computer engineering program, the theme of which is *Computer System Architectures*. One of its six PBL units, denoted APP_{comp}, targets learning of the design of compiler. As an illustration of the PBL approach, APP_{comp} is presented in detail in Sections 3 and 4.

3. COMPETENCE AND KNOWLEDGE IN APP_{comp}

In this section, we present the competencies targeted in APP_{comp}, the necessary knowledge for developing such competencies, and the documentation used as resource.

3.1 Competencies targeted in APP_{comp}

We do not consider the design of a whole compiler. We are simply interested in the design of two important components of a compiler, namely, a *lexical analyzer* and a *syntactic analyzer*. This is sufficient for learning many of the basic principles of compiler design. These two components are also commonly called *scanner* and *parser*, respectively. The following four competencies have been identified:

C1: To describe formally lexical units (LU) by using regular expressions(RE) and finite state automata (FSA)
More precisely, from a set of LU defined intuitively (e.g., textually), the targeted ability is to describe these LU using RE and FSA.

C2: To describe formally a syntax by using a grammar; to analyze and manipulate a grammar
More precisely, the goal is to be able to:
1) describe with a grammar a syntax that is initially defined intuitively (e.g., textually);
2) identify the category of a given grammar (e.g., LL(1), LR(1)); and
3) modify a grammar in order to satisfy desired constraints.

C3: To design and realize a lexical analyzer
More precisely, from a given set of RE or FSA describing accepted lexical units, the targeted ability is to construct a software that reads a text and:
1) checks whether the lexical units contained in the text are accepted;
2) returns the recognized lexical units.

C4: To design and realize a syntactical analyzer
More precisely, from a grammar describing a syntax, the targeted ability is to construct a software that reads a text (e.g., expression, program) and:
1) checks whether the text is accepted by the grammar;
2) constructs an abstract syntax tree (AST) that models the syntactic structure of the text.

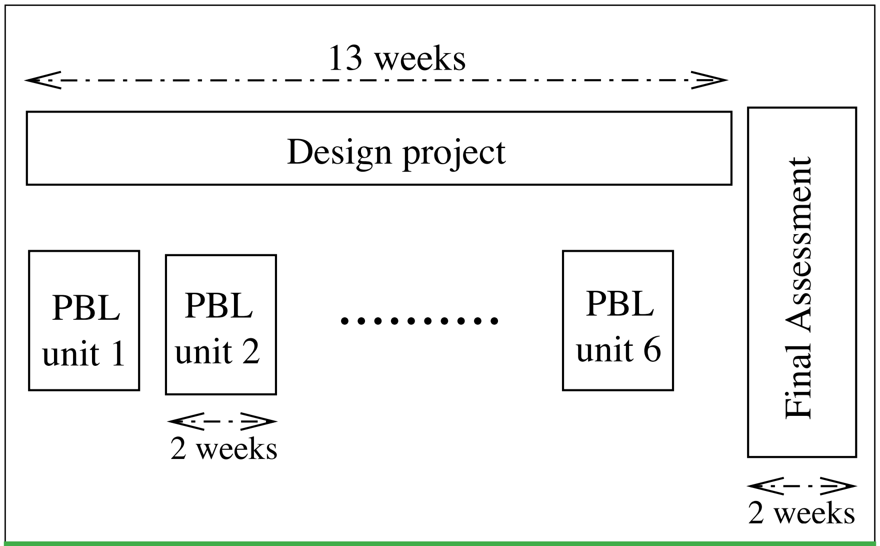


Figure 1. Semester structure.

3.2 Necessary knowledge in *APP*$_{comp}$

Development of competencies of Section 3.1 requires acquisition of knowledge, which has been classified into three types: declarative, procedural, and conditional (see Section 2.1).

3.2.1 Declarative knowledge

Declarative knowledge consists of factual information to be known. We have identified the following elements of declarative knowledge:

Mathematics:

- Regular expressions (RE)
- Context-free (or algebraic) grammars
- Abstract syntax trees (AST)
- Associativity and priorities of operators

Remark 1: Finite state automata (FSA) is declarative knowledge already acquired.

Remark 2: Trees in general are declarative knowledge already acquired. Here, we are interested in a particular type of tree called *abstract syntax tree*.

Engineering sciences:

- Lexical analysis: method based on FSA programming
- Syntactical analysis: Top-down methods, recursive-descent parsing (RDP)
- Postfix expressions

3.2.2 Procedural knowledge

Procedural knowledge is to know *how* to use factual information. We have established the following elements of procedural knowledge from the above list of declarative knowledge elements:

- Describe lexical units of a simple language using RE and FSA
- Describe the syntax of a simple language using a context-free grammar
- Modify a grammar in order to respect given constraints
- Read necessary information in a text file
- Print in a text file the information produced at each step of a program
- Derive a lexical analyzer from a FSA
- Derive a syntactical analyzer from a context-free grammar using recursive-descent parsing (RDP)
- Construct an AST
- Evaluate an AST obtained from an arithmetical expression
- Read an AST obtained from an arithmetical expression
- Derive a postfix expression from an AST
- Create validation tests of a syntactical analyzer

3.2.3 Conditional knowledge

Conditional knowledge is to know *when* and *where* to use factual information. We have identified the following list of conditional knowledge elements:

- Select a grammar that respects given constraints
- Select data structures appropriate for an AST
- Select data structures for a lexical analyzer
- Select data structures for a syntactic analyzer
- Select validation tests for a lexical analyzer
- Select validation tests for a syntactic analyzer

Note that other (declarative, procedural, conditional) knowledge is necessary, but is assumed already acquired.

3.3 Documentation

The documentation so far used in *APP*$_{comp}$ has been prepared by the first author of this article and consists of several chapters entitled:

- Introduction to languages and to compilation
- Lexical analysis, regular expressions, finite state automata
- Grammars and languages
- Abstract syntax trees
- Introduction to syntactic analysis
- Top-down parsing
- Arithmetical expressions: representations (e.g., infix, postfix, AST), operators associativity and priority

Monday–1	Tuesday–1	Wednesday–1	Thursday–1	Friday–1
Tutorial–1 (1h30)	Personal study	Problem solving procedures (3h)	Collaboration for problem solving (3h)	Project (3h)
Personal study		Laboratory work (3h)	Personal study	Personal study
		Personal study		

Week 1

Monday–2	Tuesday–2	Wednesday–2	Thursday–2	Friday–2
Problem solving procedures (3h)	Problem solving validation (3h)	Tutorial–2 (1h30)	Project (4h)	Personal study
Personal study	Personal study	Personal study	Personal study (formative assessment)	Sommative assessment (2h)
			Consultation (1h)	

Week 2

Table 1. Activities of *APP*$_{comp}$.

4. PEDAGOGICAL ACTIVITIES IN *APP*$_{comp}$

In Section 3, we presented: the objective of *APP*$_{comp}$, which is to develop certain competencies; the necessary knowledge that must be acquired for achieving this objective; and the documentation used as resource. In the present section, we present the pedagogical activities that have been elaborated to reach this objective.

Organization of activities of *APP*$_{comp}$ results from a slight adaptation of a generic organization elaborated by our department [2] and in [4]. The adaptation will be explained in the last paragraph of Section 4.2. A typical organization of *APP*$_{comp}$ is illustrated in Table 1, where grey zones are related to project activities. Let us detail *APP*$_{comp}$ activities in the following subsections. Note that in addition to several activities under supervision, students occupy the rest of their time with personal study.

4.1 Monday–1: Tutorial–1, problem to solve

For each group (comprising about 10 students), *APP*$_{comp}$ starts by a 90-minute tutorial meeting (denoted Tutorial-1). Through a collaborative work and under tutor guidance, students:

1. read the terms of the problem to solve, keep only the relevant terms, and formulate succinctly the problem;
   (30 minutes)
2. propose *solution alternatives* (i.e.,tasks for solving the problem) and, for each solution alternative, identify pertinent knowledge (acquired previously or to be acquired); *(45 minutes)*
3. organize and prioritize *solution alternatives*; *(10 minutes)*
4. review the list of knowledge to be acquired. *(5 minutes)*

The tutor's role in Tutorial-1 consists essentially in asking relevant questions, validating students' prior knowledge, ensuring that learning needs and *solution alternatives* are well identified. But the tutor never presents solutions to the problem. For the sake of clarity, let us introduce the problem that has been used this year (2004) in *APP*$_{comp}$.

4.1.1 Problem to solve

In the company where you work, a high-level programming language is used for the development of programs to be downloaded in on-board systems. A compiler has been developed in the past for the target environment used so far. For technical and economical reasons, the management of the company has decided to replace the target environment. The existent compiler must thus be abandoned and a new compiler must be designed.

In order to avoid having to design a whole compiler after each change of the target environment, your boss commissions you to design and realize the so-called frontal part of a compiler. Your boss advises you to heed Samy's advice, an experienced colleague in compilation but who has no time to realize himself the task which you are asked to do. After a first meeting with Samy and following his advice, you decide to produce a first prototype that carries out the syntactical analysis of arithmetical expressions. This is the objective considered in this PBL unit.

*Arithmetical expressions consist of operands, operators, and opening and closing parentheses. Operands are of type integer. Operators are addition (+), subtraction (-), multiplication (*) and division (/). Operator priorities are as follows: + and - have the same priority; * and / have the same priority; * and / have priorities over + and -. The four operators are right-associative. Note that priorities can be forced by using parentheses.*

An operand can have one of the following two forms:

- *Non empty sequence consisting of numbers 0 to 9.*

- *Upper-case letter followed by a (possibly empty) sequence consisting of characters such that:*
    - *each character is an upper- or lower-case letter or an underscore "_",*
    - *an underscore cannot be followed by another underscore, and cannot be the last character of an operand.*

Examples: AbcDe ZxjM_q_Sn
*Counter-examples: aBcDe ZxjM__qSn ZxjM_q_*

*Here is an example of arithmetical expression: (A_a + B_b) * C_n / 74. Operations are executed in the following order: addition, division, and multiplication.*

After a second meeting with Samy and in agreement with him, you have decided to start the design of a particular module called lexical analyzer which will be used by the syntactic analyzer. The approach, selected by Samy and your boss, consists of specifying the lexical units using regular expressions (RE) and finite state automata (FSA). A lexical analyzer can then be derived in a systematic way from the FSAs generated. It has been decided not to use Lex or any other similar software tool.

Once the lexical analyzer is terminated, you need to design the syntactical analyzer. After some research, you learn that there exist two categories of syntactical analysis methods: bottom-up methods and top-down methods. All these methods require the use of a grammar for modeling the syntax of expressions to be analyzed. In order to obtain a simple compiler whose design does not require the use of specialized software tools, the method which has been suggested to you, and which you select, is called recursive-descent parsing (RDP).

Samy informs you that RDP is applicable only if the grammar that specifies the syntax of arithmetical expressions is of a certain type and respects certain constraints. He advises you to study top-down methods, in particular LL(1) method, so that you'll be able to construct a suitable grammar.

Besides determining if an arithmetical expression is syntactically correct, a significant task of a syntactical analyzer consists of producing an abstract syntax tree (AST) that models the syntactic structure of the arithmetical expression. After construction of the AST and in order to check correctness of this construction, you must: 1) evaluate the AST, that is, calculate the result of the corresponding arithmetical expression (assuming all its operands are values); and 2) read the AST, that is, translate the data structure of the AST into a readable form (on the screen or in a text file).

In order to benefit from the advantages of object-oriented programming, you intend to use one of the two object-oriented languages that you know: C++ or Java. Since the real-time aspect is not primordial and in order to promote portability, the management has selected Java. Besides, Samy has prepared skeletons of java classes for you (provided on the Web page of APP_{comp}).

4.1.2 Results of Tutorial-1

At the end of Tutorial-1:

- The problem formulated succinctly by students must look like:

  The objective is to design and realize a syntactical analyzer of arithmetical expressions by using the method called recursive-descent parsing.

  This syntactical analyzer:
  - *calls a lexical analyzer that must be designed and realized by using finite state automata,*
  - *not only determines if an arithmetical expression is syntactically correct, but also produces an abstract syntax tree that models the syntactic structure of the arithmetical expression.*

  No speciaized software tool must be used.

- The knowledge identified as necessary for solving the problem must be close to the list of Section 3.2.
- Students must agree on an organized list of tasks (so-called *solutions alternatives*). This list must look like:

  1. Produce a lexical analyzer using an approach consisting in:
     - describing lexical units using finite state automata (FSA);
     - deriving a lexical analyzer in a systematic way from a FSA.
  2. Produce data structures that allow to:
     - construct a AST,
     - read and evaluate a AST.
  3. Produce a syntactical analyzer by using recursive-descent parsing. This analyzer must, for every given arithmetical expression E:
     - check if the syntax of E is correct,
     - construct an AST that models the syntactic structure of E.
  4. Use Java as a programming language
  5. Validate the produced software:
     - prepare a testing plan,
     - carry out tests.

     In the test of each syntactic analysis execution, you must evaluate and read the AST constructed.

     Note that *testing* is not mentioned in the problem to be solved, but students already know that every software produced must be tested.

4.2 Wednesday-1: Problem-solving procedures and laboratory work

On Wednesday-1, students have two supervised activities which are introduced in the following two subsections, respectively.

4.2.1 Problem-solving procedures

In the morning and under tutor supervision, students apply knowledge acquired in personal study, by practicing problem-solving procedures in a 3-hour session. This activity consists in solving several exercises and targets practicing of:

- describing lexical units by regular expressions (RE) and finite state automata (FSA),
- constructing a FSA that recognizes lexical units,
- deriving a lexical analyzer from a FSA,
- constructing an abstract syntax tree (AST) corresponding to a given arithmetical expression,
- using a grammar by applying corresponding production rules,
- deriving a syntactical analyzer from a grammar.

Exercises for this activity have been carefully elaborated for practicing relevant knowledge to solve Items 1-3 of the solution alternatives (Section 4.1.2). In this activity, exercises are solved by students and presented by them to their peers. The tutor validates the solutions presented, but (s)he must not present solutions him/herself.

During this session, students have the opportunity to design on paper a simple lexical analyzer and a simple syntactical analyzer.

4.2.2 Laboratory work

In the afternoon and under tutor or assistant supervision, students have a 3-hour laboratory session. They create java classes that implement the lexical and syntactical analyzers designed in the morning session of "problem-solving procedures". For each of the five java classes to be realized, a skeleton is provided to students in order to avoid their spending time on details not related to the targeted competencies. The java classes created will be completed and adapted later for the resolution of the problem. This activity aims at practicing knowledge related to Items 4-5 of the solution alternatives, in addition to continuing the practice of knowledge related to Items 1-3.

Note that in comparison to the generic organization of our department [2], we have added the constraint that the Laboratory session must occur *after* the Problem-solving procedures session. We think that this constraint improves learning because the students create in the afternoon what they have designed in the morning.

4.3 Thursday-1: Collaboration for solving the problem

Through a 3-hour session and under tutor guidance, students use knowledge acquired so far (in supervised activities and in personal study), and collaborate to elaborate solutions to the problem. After having practiced activities (through problem-solving procedures and laboratory work) of Wednesday-1, students should be able to start the resolution of the five items identified in Tutorial-1 (Section 4.1.2). The tutor's intervention consists of asking questions, making comments, drawing students' attention to relevant points, validating students' solutions, etc, but *not* presenting solutions to the problem.

4.4 Monday-2: Problem-solving procedures

Under tutor guidance, students practice problem-solving procedures in a second 3-hour session (in addition to the session of Wednesday-1 morning). This activity targets practice of:

- identifying lexical units from an informal description of a syntax,
- constructing a grammar that models a syntax defined informally,
- deriving postfix expressions.

The activity also targets continuing practice of some items of Wednesday-1:

- describing lexical units by RE and FSA,
- designing a syntactical analyzer,
- deriving AST.

After having practiced activities of Wednesday-1 and Monday-2, the students should be able to solve the entire problem.

4.5 Tuesday-2: Problem-solving validation

In a 3-hour session, students vaidate their solutions in the presence of a supervisor (tutor or assistant). More precisely, students:

- explain to the supervisor the method used to solve each item of the solution alternatives (Section 4.1.2);
- present the results obtained. More precisely, students apply their lexical and syntactical analyzers to test cases provided by the supervisor.

Some test cases are selected in order to check detection of different types of errors; for example:

Lexical errors: identifier with two consecutive underscores, identifier terminating by an underscore, identifier starting by a lower-case letter, unknown character.

Syntactical errors: consecutive operators, missing operator, missing parenthesis, missing operand.

Other test cases consist of correct arithmetical expressions with one, two or three operators, and are targeted at checking lexical unit recognition and AST construction.

The supervisor validates solutions, makes comments, draws students' attention on missing or incorrect points, but does not provide any correct method or result.

4.6 Wednesday-2: Tutorial-2

Each group of students has a second 90-minute tutorial meeting (denoted Tutorial-2). Under tutor guidance, students reflect on what they have learned, and determine if anything is missing in their understanding of the problem. By asking questions, the tutor helps students in the following steps:

Validation of knowledge acquired:
(60 minutes) Students:

- review conclusions that were generated in Tutorial-1 (see Section 4.1.2), that is: a succinct formulation of the problem, and solution alternatives;
- state the concepts that have been used in their study. The tutor makes sure that all essential concepts are reviewed, and checks if necessary knowledge (see Sect. 3.2) is acquired correctly.
- generalize and de-contextualize the new knowledge. For example:
    - o Use of FSA: modelling, design, analysis of discrete event systems (software, automation,etc).

o Use of trees: binary search trees (optimizing access time for reading, searching, writing data).

o Use of recursive procedures: computing series defined recursively, such as $S_n = S_{n-1} + S_{n-2}$.

Assessment of learning:
(30 minutes) Students:

- report on knowledge acquired and on proposed solutions. They determine among necessary knowledge elements identified in Tutorial-1 (see list of Section 3.2), those that are operational and those that require additional learning.
- discuss on their learning strategies.
- give their opinion about the learning and the atmosphere during the PBL unit.

Students also submit a written report (about 8 pages) presenting what has been learned in solving the problem.

The remaining activities are related to assessment and are presented in the next section.

5. ASSESSMENT IN APP$_{comp}$

Principles used in assessment are taken from [5, 2]. Since *APP$_{comp}$* targets the development of the four competencies introduced in Section 3.1, assessment must be elaborated carefully in such a way as to allow accurate evaluation of these competencies. A *formative* written assessment, consisting of several problems, is provided to students (at the end of Wednesday-2) with a detailed model answer for each problem. Besides, competency(ies) involved in each question of a problem are identified, indicated and weighted. By *weighted*, we mean that a number is associated to the competency for measure purpose. Students can thus: check their individual learning achievement by comparing their answers to the model answer, measure to which level each competency is developed, and evaluate their preparation to the written exams (see below). In order to determine if a student passes or fails *APP$_{comp}$*, (s)he will be evaluated through:

- the report submitted during Tutorial-2 (see Section 4.6), which presents clearly what has been learned in solving the problem. This is a final and written report of the activities of Tuesday-2.
- two written exams, referred to as *summative* assessments, at the end of *APP$_{comp}$* (i.e., Friday 2) and at the end of the semester, respectively.

Similarly to formative assessment, competency(ies) involved in each question in a summative assessment, are identified, indicated and weighted. So far, assessment in *APP$_{comp}$* has been distributed among the four competencies as follows: C_1: 15 % C_2: 20 % C_3: 25 % C_4: 40 %

To determine if a student passes *APP$_{comp}$*, let us consider the three possible situations:

1. A student passes *APP$_{comp}$* if (s)he is evaluated at least 50 % for each of the four targeted competencies.
2. A student fails a competency if (s)he is evaluated below 50 % in this competency but gets at least 50 % globally (i.e., average over the four competencies is at least 50 %). The student must be assessed again on the failed competencies through a written exam. The student and the tutor agree on a date for the exam within a period of one semester after the end of the unit. The student passes *APP$_{comp}$* if (s)he obtains at least 50 % in every competency evaluated in the exam. Otherwise, the student fails *APP$_{comp}$* and (s)he is in Situation 3.

It is worth noting that if the student passes the exam (and thus *APP$_{comp}$*), (s)he is evaluated exactly 50 % in the competencies of the exam (i.e., competencies initially failed) even if (s)he obtains more than 50 %.

3. A student fails *APP$_{comp}$* if (s)he is evaluated below 50 % globally. The student must register to the unit the next year, like a student that takes *APP$_{comp}$* for the first time. This induces a translation in the student's schooling because, by taking *APP$_{comp}$* a second time, the student will have to postpone another unit. This translation may be propagated during several semesters, possibly until (s)he terminates his/her bachelor's degree.

No documentation is allowed during summative assessments.

Note that this PBL unit is worth 2 credits, with a total of 15 credits for the semester, and a total of 120 credits for the whole computer engineering program.

6. DISCUSSION

Let us compare the PBL approach to "conventional" courses, in the context of APP_{comp}. With the PBL approach, students are *sooner* and *better* prepared for designing lexical and syntactical analyzers and for developing java programs with recursion. Let us clarify what we mean by *sooner* and *better*.

6.1 Students are prepared sooner

APP_{comp} takes place during Semester 3, and its main learning objectives can be categorized as follows:

- Design of a lexical analyzer,
- Design of a syntactical analyzer,
- Programming in Java with recursion.

In our previous (course-based) computer engineering programs, students had to wait until:

Semester 4 for studying Item 1 and partially Item 3, in a course entitled *Data structures and algorithms* (DSA). We say "partially" about Item 3, because students were not forced to use *recursion*.

Semester 7 for studying Item 2, in a course entitled *Language organization and compilation* (LOC).

6.2 Students are better prepared

With the new programs, learning *contextualization* is promoted by permitting the students to solve real problems at the early stage of their learning. In APP_{comp}, the problem to solve is simple and a little amount of code is developed, but students need to think about and master concepts before coding. For example, in a laboratory session of APP_{comp}, students practice knowledge related to the resolution of a concrete problem (i.e., to the design of a compiler). This approach is used starting with the first semester of our new programs. During semesters 1 to 6 of the traditional programs, laboratory assignments were mainly oriented on a subject (e.g., automata) instead of on problem resolution. Students had to wait until Semester 7 to have some (and not all) laboratory assignments aimed at problem resolution.

Another ability developed with the new programs is *integration*. In APP_{comp}, students integrate several concepts that were studied separately in the course DSA. They also integrate several concepts of the courses DSA and LOC.

Another advantage of the new programs is that during a two-week period, students use on average 80 % of their time in studying concepts related to the current unit (e.g., APP_{comp}), and thus, to the resolution of a single problem. The remaining 20 % of the time are used for the project that lasts the whole semester. In standard practice (i.e., previous programs), students study in parallel five different courses the whole semester, which implies frequent awkward "switchings" between subjects very distant with each other. For example, in previous programs, students may have had as many as five independent laboratory assignments during the same week.

Let us also note that in the course LOC of our previous programs, students used specialized software tools (LEX and YACC) for generating their lexical and syntactical analyzers. They spent much time in learning how to use these tools, to the detriment of learning concepts directly related to the three items mentioned above (in Sect. 6.1).

Regularly, at the end of each PBL unit, the person responsible for the semester has a meeting with students in order to receive their comments and feedback about the unit. At the end of the semester, supervisors involved in PBL units or in the project, have a meeting with students in order to receive their global comments. Student feedback has been positive for the semester; and in particular for APP_{comp}, it has been very encouraging in many aspects, such as their learning and interest, and their appreciation of tutors and assistants. Interestingly, we have heard from supervisors involved in industrial training semesters that students coming from the new programs are greatly appreciated.

7. Conclusion

The Department of Electrical and Computer Engineering of the *Université de Sherbrooke* has undertaken a major reform of its programs. The new pedagogical approach is based on *competence development* for *solving problems* and *realizing design projects*. As an illustration of the problem-based learning (PBL) approach, we present a two-week PBL unit that targets developing and assessing competencies related to compiler design. Student feedback has been very positive and encouraging about their learning, interest and their appreciation of supervisors.

Note that APP_{comp} is not a purely practical unit, but it is also based on several theoretical subjects (automata, grammar, ...). Thus, with this study, we have also demonstrated that PBL can be applied to non-purely practical subjects.

Acknowledgements

Many thanks to Prof. Frédéric Mailhot who has proofread the article and improved the English significantly.

References

[1] Canadian Academy of Engineering. Evolution of Engineering Education in Canada, 1999. http://www.acad-eng-gen.ca/publis/publi_an.html.

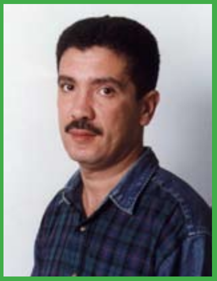
[2] G. Lachiver, D. Dalle, N.Boutin, A. Clavet, F. Michaud, and J.-M. Dirand. Competency- and Project-Based Programs in Electrical & Computer Engineering at the Université de Sherbrooke. *IEEE Canadian Review*, pages 21-24, summer 2002.

[3] A. Khoumsi and B. Hadjou. Learning Probabilities in Computer Engineering by Using a Competency-and Problem-Based Approach *the Jour nal of STEM Education: Innovations and Research*, Vol. 6, Issues 3&4, July-December 2005.

[4] D. Woods. Problem-Based Learning: resources to gain the most from PBL, 1996.http://chemeng mcmaster.ca/pbl/append-a.htm.

[5] M.E. Huba and J.E. Freed. *Learner-Centered Assessment on College Campuses. Shifting the Focus from Teaching to Learning*. Boston: Allyn and Bacon, 2000.

Ahmed Khoumsi received the Engineer degree in Aeronautics and Automation from the engineer school SUP'AERO (Toulouse, France), in 1984. From 1984 to 1988, he achieved his research activities in the LAAS, a CNRS research center, in Toulouse. In 1988, he received the Ph.D. degree in Robotics and Automation from the University Paul Sabatier in Toulouse. From 1989 to 1992, he was an Assistant Professor in Robotics and Computer Engineering at the engineer school ENSEM (Casablanca, Morocco). From 1993 to 1996, he was a Postdoctoral Fellow in the Communication Protocols group of the University of Montreal. From 1996 to June 2000 he was an Assistant Professor, and since then he is an Associate Professor in the Department of Electrical and Computer Engineering, at the University of Sherbrooke, in Canada. His present research activities include: modeling, testing, and control of distributed and real-time systems; design and creation of telecommunications services; and application of new learning methods in teaching.



Ruben Gonzalez-Rubio received the Engineer degree in Electronic and Communications from the engineer IPN ESIME (Mexico City, Mexico), in 1972. From 1972 to 1978, he worked in industry. In 1981, he received the Ph.D. degree in Computer Engineering from the ENST (École Nationale Supérieure de Télécommmunications), Paris, France. From 1981 to 1988, he was with the BULL research center, in Paris. In 1987, he received the Doctorat ès Sciences degree in Computer Engineering and Mathematics from the Pierre et Marie Curie University, Paris, France. From 1989 to now, he is a Professor in the Department of Electrical and Computer Engineering, at the University of Sherbrooke, in Canada. His present research activities include: all aspects of software engineering, optimisation, build software applications.