Learning compiler design as a research activity

Francisco Moreno-Seco, Mikel L. Forcada

Departament de Llenguatges i Sistemes Informàtics,
Universitat d'Alacant,
E-03071 Alacant, Spain.
E-mail: {paco,mlf}@dlsi.ua.es

May 27, 1996

Running Head: Learning compiler design as a research activity

Abstract

This paper describes the application of a pedagogical model called "learning as a research activity" [D. Gil-Pérez and J. Carrascosa-Alis, *Science Education* **78** (1994) 301–315] to the design and implementation of a two-semester course on compiler design for Computer Engineering students. In the new model, the classical pattern of classroom activity based mainly on one-way knowledge transmission/reception of pre-elaborated concepts is replaced by an active working environment that resembles that of a group of novice researchers under the supervision of an expert. The new model, rooted in the now commonly-accepted constructivist postulates, strives for meaningful acquisition of fundamental concepts through problem solving —in close parallelism to the construction of scientific knowledge through history.

1 Introduction

This paper describes the implementation of an adapted version of the *learning as a research activity* model ([1, 2, 3, 4, 5]) to the subject of compiler design at the University of Alacant. The need for trying a new teaching model arose two years ago when we confirmed our realization that the combination of classical lecturing of theory and solved problems in the classroom and open laboratory assignments was insufficient for most students to learn the basic skills needed to successfully tackle new problems in the field. The *learning as a research activity* model, which has been successfully applied to the teaching of science in secondary school, was adopted, adapted, and applied to our subject. We have found the new model to be more adequate, more successful, and more consistent with an engineering education style.

The classical model used in most universities in Spain to teach computer engineering subjects such as compiler design could be simplified (in its extreme form) as follows. In the classroom, the teacher tries his or her best to explain carefully the theoretical and conceptual aspects of the subject, usually assuming —but seldom carefully checking— that the students are familiar with a set of basic concepts on which the new material is based. After that, some more lecturing time is devoted to showing, with as much detail as possible, the teacher's or a book's solution to a handful of selected (intendedly *representative*) problems or exercises. Laboratory work may be organized either around an open or closed structure, but usually consists either in building one or more programs that implement the algorithms and techniques taught in the classroom to solve a set of selected problems or in writing applications according to some specified requirements. In addition, the teacher is available during some scheduled office hours to assist students with their personal work.

Of course, significant deviations from this simplified model are observed: a few students may interact with the teacher during an explanation, usually to ask him to repeat something, or explain some passage better, or to point out a minor blunder; teachers may ask questions to the students to try to make sure that they are following the explanation; the teacher's solution to a given problem may only be explained after students have had some time to think about it, etc.

Our experience shows that the average performance of our students when we taught the subject according to this model was usually low when they were asked to solve new, open problems —a situation which will be common in a computer engineer's professional practice— during, for example, written tests. When the proposed problems depart from those whose solution was explained in the classroom, a large fraction of students fail, either because the concepts involved have not been properly understood or applied or because they are unable to design a strategy to deal with a previously unseen situation[1]. In some cases, the fraction of the studentship with passing grades is higher than would be reasonable to expect in view of these facts. This is due to a lowering of standards during grading, especially when teachers acknowledge that students have been working hard anyway.

Our early teaching practice was not very different from this charicature and its results were quite similar: we decided to give passing grades to students who had clearly failed in spite of having worked really hard. Since we had been appointed to educate future engineers who should be successful when facing new problems in their computing profession, we decided to reexamine our teaching practice. The *learning as a research activity* model was initially considered mainly because of its emphasis in problem solving.

The next section outlines the main features of the *learning as a research activity* model. Section 3 describes the particular situation in which compiler design is taught in our University. The application of the new model to our subject is explained in section 4. Future developments and challenges are discussed in section 5. A course outline and a sample of classroom material (translated from Spanish) are given in appendices A and B.

2 Learning as a research activity

This section will present a brief summary of the *learning as a research activity* model. The reader interested in more details is referred to [1, 2, 3, 4, 5] Readers who are familiar with Spanish are also referred to the book by Gil *et al.* [6].

2.1 Constructivism in science education

The *learning as a research activity* model is based on a *constructivist* approach. Constructivism is widely considered to be a very important contribution to science education

[1]This happens partly because solving problems is not an algorithmic process; teachers hide the trouble they have had when solving problems and teach their solutions in the classroom instead of teaching how they faced the task of solving the problems.

(see [2, 7]). Here is what constructivists say about the learning process:

- Learning is the construction of understanding. Understanding goes beyond mere retrieval and is based on establishing relationships between concepts.

- If information items appear in isolation, they are easy to be forgotten or to become inaccessible.

- All learning depends on prior knowledge. New concepts are learned in relationship to previously learned concepts.

A pedagogical model motivated by constructivism has to take this description of the learning process into account when designing classroom activity. As a consequence, a constructivist approach to teaching relies on a careful analysis and elicitation of concepts and ideas that may exist prior to learning. New concepts are anchored in those preconceptions that may be useful; preconceptions that interfere with learning are challenged.The newly learned concepts are finally applied to new situations. The whole process is usually called *conceptual change strategy*. In areas where misconceptions abound, conceptual change has to be used carefully to avoid inhibiting student participation ("why say anything, we're always wrong").

2.2 A methodological change

The new model goes one step beyond constructivism: in addition to conceptual change, a *methodological change* is proposed. This is based on the hypothesis that misconceptions (analysed mainly in connection with physics and chemistry in the work of Gil-Pérez and Carrascosa-Alis [2]) are due to a "methodology of superficiality", or "common sense", that pervades everyday thinking and lies quite far from the ways of science (and technology, we should add). The main focus of the new method is on "problem solving". Science and technology have advanced by creating knowledge, inventing new concepts and revising old ones in response to new problems; it is proposed that a similar methodology could be applied in the classroom. Concepts and theories should appear only after the problems that motivated them are clearly stated and assumed by students. Classroom activity should encourage creative, divergent thinking and a rigorous process of theoretical and experimental hypothesis checking

—the very ingredients of scientific and technological advance— towards the construction of consistent bodies of knowledge.

2.3 A metaphor: students as novice researchers

The resulting teaching/learning model is structured around the metaphor that sees students as novice researchers (instead of passive receivers of previously elaborated knowledge), who perform their research and construct their knowledge of the subject under the supervision of an expert researcher, the teacher[2]. Classroom activity is structured around cooperative work in small groups that later interact and confront their findings with those of the scientific community (represented by the teacher and the books). A course is driven by a carefully designed sequence of open problematic situations that motivate the students both to attack them and to construct the concepts and techniques of the discipline.

It may be argued that it is impossible that students construct the knowledge of a whole discipline by themselves. But the model does not expect them to do so either. Indeed, a novice researcher (e.g., a Ph.D. student) does not construct the whole field he or she is working in by himself or herself, but the dynamics of the research group allow the newcomer to successfully and rapidly "catch up" through work and communication with other researchers in the group. Even if students do not completely solve a posed problem, their attempt and their advances (the study of the problem, the hypothesis they have formulated, the strategies they have outlined) prepare them to meaningfully learn the concepts and theory involved in its solution, because they become aware of "where the problem is". The information provided by the teacher or by other groups becomes more relevant because it refers to problems that students have previously studied and tried to solve.

[2] Abboud [8] notes (our italics):

> In the traditional method, the teacher structures the knowledge [...] and presents it to the students. The teacher is the actor and the students are the recipients. The approach here [*in her programming course*] is more student-centered, in that the student is an active participant in the learning process. The teacher provides *guidance and a stimulating and safe environment* [...]. Learning becomes [...] the mechanism for the creation of new knowledge [...]. The classroom becomes a laboratory, where the student joins actively in the analysis, hypothesis formulation, and verification processes. [...] In this view, the learning becomes *experiencing this process, rather than getting the end result.*

2.4 Classroom activity

At the beginning of each block (group of sessions) the teacher provides the students with a list of activities (small questions, open problems, etc.). These activities are carefully designed to "anchor" the new concepts to previous ones by working on representative problems. The classroom is organized in small groups (of about four students) that work each one of the programmed activities for a while. When the teacher decides that the groups have advanced enough on their way to the solution (through monitoring of their work), a "discussion in common", moderated by the teacher, takes place. This discussion, in which the representatives of one or two groups explain their solution and discuss them with the rest of the class, allows the teacher to summarize, correct, and synthesize the students' proposals, add new information, and prepare them for the next activity.

For the reasons given above, it is not necessary to wait until all groups complete the activity —it may even be almost impossible due to the nature of the particular problem—; instead, the teacher interrupts the work of the small groups and starts the discussion in common when he or she thinks that the students are ready to understand the solution given to the problem by the scientific or technical community. The exact moment of this interruption depends on the activity which is being solved; sometimes this moment can be established while designing the activity, at other times the teacher has to decide when to interrupt using his or her intuition, depending on the pace of the class. Our experience shows that this last situation is more frequently observed, maybe because we design the activities so that they can be completed succesfully by our students (except for a reduced number of activities which we know to be very difficult for our students to solve them completely).

When applying the new pedagogical model a number of difficulties may appear (see [6]):

- Active learning situations usually require more time to "cover" the program of the subject than traditional approaches do. However, the success of encyclopedic syllabi is limited due to shallow and confuse learning on the part of students. When designing the program of activities, the teacher has to focus on the meaningful acquisition of the key concepts of the subject, with an emphasis on the creation of a *coherent* body of knowledge.

- Sometimes, the pace of the class makes the program of activities inadequate or calls for a new activity to be inserted. Other times, an activity fails to introduce a concept because of an inadequate design. The teacher has to be ready to improvise and adapt the program to the progress of a given class, and that requires considerable skill and experience.

- Students sometimes choose not to make the effort of working the activities out and may decide to simply listen from the teacher. The new model requires a special ability from the teacher, with a metacognitive goal: one has to *convince* students that their effort pays, because then they will understand much better what the groups discuss and what the teacher explains at the end of each activity[3].

- Some students may get disoriented by the new style, and learn in an incoherent way. The 'closing explanation' at the end of each activity by the teacher is crucial in connection with this difficulty.

This difficulties have to be seriously considered; hence the necessity of continuously evaluating not only the students' learning, but also their attitude and the performance of the program of activities designed.

2.5 The role of evaluation

Everyone agrees that evaluation has a central role in any teaching model. Students usually consider it to be one of the most important issues, because the way they will be graded greatly influences the way they will organize their work; students will focus mainly on those issues that will be taken into account when grading. Therefore, any pedagogical change is incomplete without a parallel change in grading practices.

The teacher's expectations on a given group of students are also critical to their learning process. If the teacher assumes that a certain percentage of the group is bound to fail, this will affect his or her teaching in such a way that some students will probably consider themselves to be part of that percentage: the result is a self-fulfilling prophecy. On the other hand, a

[3]This can be accomplished (in a constructivist way) by means of a continued positive reinforcement by the teacher.

teacher that expects all students to succeed will transmit this encouragement through his or her ways of teaching.

The *learning as a research activity* model assigns evaluation a radical role, well beyond equating evaluation to a mere measurement of achievement. Evaluation

- is continuous to maximize its influence in the learning process throughout the course;
- anticipates obstacles and helps students to eventually overcome them;
- gives students an opportunity to know their achievements and their needs, that is, acts as a positive reinforcement toward improving their learning process;
- informs students about their degree of accomplishment in a set of well-defined objectives;
- considers methodological issues (the "how") in addition to the acquisition of concepts (the "what") —this should be central to any learning process structured around problem solving—;
- includes the classroom atmosphere, the teacher's attitude, etc.

An important issue affecting evaluation is the pervasiveness of what Novak [7] calls *positivism*. In positivist teaching practice, the only source of evaluation is the teacher: a solution or an idea is *good* or *bad* if the teacher says so, independently of the fact that it may or may not advance in the solution of the proposed problem(s) in a consistent way. Students are used to years of positivist teaching practice, and tend to show their work to the teacher and still ask him or her whether it is "right" or "wrong", when exposed to a constructivist learning environment for the first time. In the new model, students are encouraged to develop an attitude towards self-evaluation. They learn to check their own solutions for consistency and to devise techniques to perform these checks.

2.6 Syllabus design as a research activity

When using classical pedagogical schemes, the syllabus is designed around a sequence of lectures that covers the contents of the course. Practical work (homework assignments, laboratory assignments, and possibly laboratory projects) usually *follow* the lectures. With this new model, lectures are substituted by a program of activities designed to encourage students to construct and learn by themselves (with the necessary help from the teacher) the basic concepts of the subject. The design of this program of activities should be made according to the students' previous knowledge[4] and to their ability to solve related problems. When the students have learned the subject, they work on a special set of *application activities* designed to apply the newly learned concepts to new problems. These activities are different from homework assignments in that the students solve them mostly during classroom sessions, discussing in small groups and then discussing their solutions with other groups and with the teacher.

With this new model, the "number of concepts" to be learned during the course usually has to be reduced; however, the concepts are learned by the students in a more coherent, more interrelated way with this model than with the traditional one.

The design of the program of activities becomes a research activity itself. The program has to be always open to reelaboration or even complete restructuring; its current form is just the best try, based on hypotheses that are still to be confirmed or rejected through careful classroom experimentation. The whole learning environment is pervaded by a recurrent reference to research as a key activity: students are expected to behave as novice researchers in the classroom, but teachers also become full-time education researchers. This approach may motivate teachers to escape from the dull routine of having to teach the same subject again and again and to look at teaching in a whole new light [2]. The connection between research and teaching becomes especially important in a university environment, where teachers are usually expected to perform in both areas.

3 Course and Audience

3.1 Students' background and course organization

Compiler design is taught during the third year of the five-year Computer Engineering curriculum of the Polytechnic School of the University of Alacant. The subject is organized in two separately-credited one-semester subjects, Compilers I and Compilers II; the divi-

[4]One of the hardest tasks for the teacher is to explore and discover the previous concepts (right or wrong) the students may have (see section 4.1).

sion is mainly due to issues more related to University policy rather than to pedagogical considerations. The subjects are also electives for students pursuing a three-year degree in Computing.

The ACM/IEEE-CS Joint Curriculum Task Force Report [9, 10] defines an advanced course on Programming Language Translation (sometimes called Compiler Design or Compiler Construction) which corresponds very closely with the compiler design subjects taught at the University of Alacant[5]. The prerequisites stated in the report are similar to those established at the University of Alacant. The main difference is that the report proposes as prerequisites the basic knowledge units of Abstract Data Types (AL2), Recursive Algorithms (AL3), Complexity Analysis (AL4) and The Software Development Process (SE2) which are not considered prerequisites for Compiler Design at the University of Alacant. The fact that some of these subjects (AL2–4) are taught during the second year contributes to overcome this drawback in our curriculum, but since subjects related to software engineering are taught during the third and fourth years, our students know little about software design and specification when they attend the compiler design course. Also, the lack of a general subject on programming languages[6] is an important shortcoming in our curriculum[7].

Third-year students are supposed to have a solid background[8] in the following subjects[9].

- Discrete mathematics (AL2)
- Assembly Level Machine Organization (AR4)
- Languages, Grammars and Automata (PL7, PL8)
- Turing machines, decidability and recursive function theory (AL7)

[5]There is also a basic knowledge unit, Language Translation Systems (PL9), which has an introductory purpose and hence is very limited in time.

[6]There is only a course on programming at a very introductory level which does not deal with programming languages in general.

[7]Compilers I and Compilers II are the only mandatory subjects that deal with programming languages and their processors (interpreters, compilers) in a general way, but not only in our University: an equivalent situation occurs in most other universities in Spain. The whole situation places an enormous burden on the subjects, which become more important to future engineers than its names would in principle suggest.

[8]In our University, there is a complex system of course requirements, that may be summarized as follows: students are required to get a "pass" or higher degree in six subjects to be *eligible for grading*, but not to *enroll*, in Compilers I and II.

[9]We will strongly encourage a modification of our Computer Engineering curriculum to include the prerequisites stated in the ACM/IEEE-CS report [9].

Students meet weekly in two one-hour classroom sessions and in a single one-hour laboratory session. Each of the semesters is assigned 15 weeks. In course 1994–1995, 320 students enrolled initially, but roughly 220 followed the courses regularly. There were three classroom groups and ten laboratory groups, with approximately 70 students per classroom group and 35 students per laboratory group (there were four groups with 10 or less students). These large student numbers are common in Spanish universities and become a serious challenge to any pedagogical innovation enterprise.

3.2 Students' behavior and positivism

Most of the problems we encounter when working with our students are due to their extended exposure to positivist learning, teaching, and grading practices (see the discussion in section 2.5). Indeed, current practices may date back to the Middle Ages, when the availability of books was limited and the role of lecturers was mainly to dictate knowledge and that of students to copy and learn this knowledge. This model of classroom dynamics is still alive seven centuries later, even after the Gutenberg revolution and other technological advances. The following list includes some preliminary observations about our students that most of our colleagues would accept as basically correct:

- Students go to the classroom sessions to take verbatim notes of whatever the teacher says, and use these notes as their main study material. Most of them never take the time to check their notes against what is found in manuals. Students seem to think that the teacher expects them to uncritically regurgitate what they have been told as exactly as they can (perhaps some of their teachers did), in radical agreement with positivist teaching practice (see section 2.5).

- Accordingly, students are not used to thinking, that is, to applying their own knowledge to situations that depart slightly from the examples discussed in the classroom. If a test contains a simple but new problem that may be solved using the theory they have tried to memorize, the most common result is that they fail to solve it, as explained in section 1.

- Students are very passive in the classroom. They do not ask many questions, except for "could you repeat, please?". If the teacher asks a question to the class, all he or she gets is silence; if the teacher asks a particular student, the usual answer is "don't know".

3.3 Students as programmers

We find that, even after two years of instruction, our students are in general, simply said, bad programmers. This shows clearly in our compiler design courses: we often encounter students that seem to have acquired the main concepts and learned the basic techniques of compiler construction but fail to build correct programs because of bad programming habits and serious misconceptions about programming in general and the C programming environment in particular[10]. We are unaware of the extent of this problem in other universities, but it interferes strongly with our teaching practice. Some common problems include:

- Our students know very little about *programming in the large*[11], and even less about programming in a way that eases maintenance of a large software product. Our compiler assignments are probably some of the more complex programming projects they face during their five-year curriculum[12].

- They do not even know how to test systematically their own programs or how to organize program modules so that they may be tested separately.

- Our students love using pointers and dynamic memory but do it in very unsafe ways (only compatible with single-user environments such as DOS), usually resulting in an infamous `memory fault` run-time error when the teacher tests their compilers (in a multi-user environment such as Unix). This aspect is intimately related to the previous one.

[10]For example, a great fraction of our students do not know that `getc` and `fgetc` in `stdio.h` return `int`'s instead of `char`'s. This causes serious problems when dealing, for example, with the end-of-file condition.

[11]Making a medium-sized program may involve several months or years and requires some planning and a lot of encapsulation (that is, one must design a module to be robust and encapsulated in order to use it several months later, when one does not remember the little details and tricks one was aware of while implementing it), and also requires a lot of separate module testing.

[12]Excepting, perhaps, their final-year programming project.

- They tend to "overimplement" the data structures they use in their compilers, mainly because they try to use as much as possible of the code they have generated when studying Abstract Data Types (C++ classes, etc.). Also, they tend to use code they have previously written, regardless of whether it is needed or not. Their reliance on not-thoroughly tested code sometimes results in the failure of their compiler-design assignments.

- Most of them have computers at home where they use DOS-based C and C++ programming environments that depart slightly or allow extensions with respect to standard C and C++. Their compiler assignments, therefore, fail to compile or to run properly in other environments.

- Students tend to produce compilers without having a hypothetical user in mind; instead, their compilers use "classroom jargon" instead of a more user-oriented language, for example, in error messages ("`lparen found, relop or addop expected`"). This is related to their exposure to positivist (see section 2.5) teaching practices: they are programming with the *teacher* in mind!

4 Learning/teaching compiler design as a research activity

4.1 Constructivism in teaching compiler design: preconceptions

In constructivist practice, one of the main tasks in the teacher's work is the identification of preconceptions that may have an influence in the learning process. Some of them may be correct, and therefore may be used as anchors to new concepts [11]; some others are misconceptions that may seriously interfere with the construction of new concepts if they remain undetected and are not dealt with adequately.

4.1.1 On the persistence of misconceptions

As in other fields, we have found, however, that some misconceptions are very persistent: even when they are detected and classroom and laboratory activity is designed to address them before they interfere with the progress learning, they may reappear. An example that we have found is that students think that left factoring and elimination of left recursion are enough to transform any grammar into an LL(1) grammar, just a few months after one of the laboratory assignments involved a grammar for which this did not work. A possible source for this misconception may be the incorrect reception of a correct concept: no LL(1) grammar may have left recursion or left common factors.

To make matters worse, we have found that compiler design textbooks do not seem to take the possibility of this confusion seriously enough. Almost all books we have checked ([12, 13, 14, 15]) only make emphasis on eliminating ambiguity, left-recursion and (not all books) common prefixes. Only Fischer and LeBlanc ([16], p. 126) and Aho, Sethi and Ullman ([17], p. 191) give an example of a grammar which has no left recursion and no common prefixes but it is not LL(1), but in the case of [17] they give an ambiguous grammar, which can also reinforce another misconception (see section 4.1.5).

4.1.2 On programming as an *art* (another preconception indeed)

In constrast with, say, aerospace engineering students, computer science students can build and "test fly" their designs easily, even at home. This may be at the basis of a serious preconception [18], according to which:

- Programming is an art rather than a science.
- One does not need a rigorous planning of the strategy used to solve a problem before one tries to make a program to solve it (the "design at the keyboard" syndrome).
- A good program is just a program that works for the person who wrote it (and perhaps for the user), rather than a program that is readable, modular, extensible, etc.

4.1.3 Compile-time vs. execution-time

A common source of serious misconceptions regarding compiler design may be summarized by saying that our students have trouble distinguishing *compile time* and *execution time*[13]. This may sound too vague, but we are currently working on identifying elementary preconceptions

[13] When writting syntax-directed translation schemes, a significative fraction of our students used an attribute to evaluate the *value* of each expression (they only had to evaluate its type and address), that is, they were executing the program while compiling it!

that may be at the basis of this confusion, which we have detected only very recently as one of the key sources of interference with the students' learning process. We hypothesize that the confusion arises from the widespread use by students of integrated development environments such as Borland C, where edition, compilation, execution, and debugging occur all in the same box (in these environments, compilation, or even the whole compiling–linking–debugged execution process is launched by pressing a hotkey, just as if it were another editor command). Another possible cause is the fact that, in most of their experience, compilers run on the same platform (machine, OS) as the objects they produce.

4.1.4 "Compilers first check syntax, then generate code"

Another preconception that we have detected may be formulated as follows: "compilers initially check the syntax of the user's program and, only if the program is correct, they go ahead and compile it". This may not seem a misconception as stated, but let us look at it more carefully. It may collide with the key concept of *syntax-directed translation*, that is, the idea that translation is driven by parsing or by a syntactic representation of the program generated by a parser: students may think that parsing is just a consistency check that has to be done to avoid problems during a subsequent translation process that does not rely on syntax at all[14].

We have found that this misconception, even though it may seem unharmful, may weaken the motivation of students when learning how to build parsers. The study of parsers is an important and heavily technical part —consider LR parsing, for example— of most courses on compiler design, and may be painful to learn outside the motivating frame provided by the concept of syntax-directed translation.

Indeed, most textbooks do not seem to consider how powerful it can be to make the concept of syntax-directed translation clear even before parsers are discussed. They just *reveal* the block diagram of the compiler at the beginning of the book and dive tens or even hundreds of pages deep into the description of the first two boxes in that diagram, the scanner and the parser, without a clear motivation for the reader. We have designed our classroom activities so that the concept that "obtaining a syntactic representation is

[14] Aho, Sethi and Ullman in their book ([17], p. 12) state: "*After* syntax and semantic analysis, some compilers generate an explicit intermediate representation of the source program" (our italics).

crucial to translation" appears very early in the course (just after studying the importance of syntax as a tool to *specify a language*). Students tackle the problem of translating Pascal-like arithmetic expressions to sequences of key presses for a simple calculator; they realize that an easy way to do this is to either fully parenthesize the expression in advance or, what is equivalent, to obtain a tree-like representation for the expression, so that evaluation order (precedence, associativity) is clear before translation is attempted (see appendix B).

A related issue is that when textbooks give a block diagram of a compiler, dividing it in a number of phases or passes, not all of the textbooks make emphasis on the very important question that this is a *conceptual division* rather than an *actual division*, that is, that real compilers are usually *not* divided in a number of separate phases and routines but instead consist of *coroutines* (see [15, 19]).

4.1.5 Ambiguity

Another concept that students learn incorrectly is *ambiguity*. They insist in using this concept to refer to LL(1) or SLR(1) parser conflicts caused by perfectly unambiguous grammars that are not LL(1) or SLR(1)[15] . This may be due to the fact that parser conflicts are usually illustrated by feeding ambiguous grammars into the algorithms that construct the parsing tables (as in Aho, Sethi and Ullman [17], p. 191; see section 4.1.1). We have tried to be careful when using examples, so that grammar ambiguity and parsing conflicts are presented as orthogonally as possible.

4.1.6 Conclusions on preconceptions

These are just examples of the kind of preconceptions that, if left undetected, may seriously compromise the success of a teaching strategy. A more systematic study of the elementary misconceptions interfering with the learning of compiler design concepts is underway, and will be published elsewhere.

[15]They seem to confuse *ambiguity* (applied to grammars) with *nondeterminism* (applied to parsers).

4.2 Classroom sessions in a compiler design course

We applied the techniques described in section 2.4 to our compiler design course, and we found that our students were able to solve —almost by themselves— the following problems[16]:

- enumerating the tasks a compiler performs when processing a program;
- designing an algorithm to decide whether a given string belongs to the language specified by a grammar or not ;
- translating an arithmetic expression to a sequence of keypresses for a pocket calculator;
- designing a grammar for simple expressions with numbers, parentheses and arithmetic operators ($+$,$-$,$*$,/), with the operator associativity and precedence of programming languages like Pascal or C;
- designing an algorithm to obtain the initial symbol of the grammar from the input string, following the inverse of a rightmost derivation.

With this method, our students were capable of learning the basics of lexical analysis, recursive-descent parsing, LL(1) parsing and intuitive shift-reduce parsing; the last concept was used as a sound basis on which students were taught LR parsing and SLR parser construction[17]. We have found that our students understood bottom-up parsing better than students from previous years for two reasons: they had deeply examined the problem to solve, and they were highly motivated to learn the solution.

The subject of syntax-directed translation was also learned by our students using this method: we began with an implementation-oriented strategy (it was very easy for the students to insert translation actions into a recursive-descent parser), and then we inferred from them the concepts of syntax-directed translation scheme (SDTS) and syntax-directed definition or attributed grammar. Our students also learned to do type checking, to generate code for

[16]See appendix B for a more detailed formulation and motivation of some of these problems. All of the examples in this section belong to Compilers I.

[17]The construction of canonical LR(1) and LALR(1) tables was not covered in our course because we think that a detailed study of these subjects is not necessary: a student that knows *and understands* how to construct a SLR(1) table can learn —by taking a look at a book— those complex constructions by himself or herself (very few people have made a LR(1) table or a LALR(1) table for a medium-sized language by hand).

expressions with arrays and records, to translate a small subset of a Pascal-like language to code for a stack machine, and to generate code for a virtual machine very similar to three-address code. Finally, the translation of procedures and functions was discussed and worked thoroughly, and concepts such as that of activation record were studied (again, they were not capable of finding the solution to function call compilation by themselves, but after having worked hard on the problem they were very well prepared to "receive" it). As we will see in the next section, we asked our students to write three small toy compilers in their laboratory sessions. And, even though we did not explain anything about compilation of Pascal-like procedures and functions (with procedures and functions declared inside them and nested scoping rules), some of our best students were able to write compilers that dealt with them (we had some interaction with these students, but almost all of the work was theirs).

4.3 Laboratory sessions

In parallel to classroom sessions, the students had to write several programs to pass the course. These programs had three main objectives: to apply the concepts learned in the classroom sessions; to study in depth the problems studied in the classroom sessions; and to have the students build a medium-sized software project.

During year 1995–1996, our students will have written:

1. a small program to deal with the end-of-file condition in different operating systems
2. a general parser for context-free grammars[18] (even for ambiguous grammars)
3. a recursive-descent parser
4. a LL(1) parser
5. a SLR(1) parser
6. a small compiler for a toy language and a stack machine
7. a compiler for a simple language with control structures, on the same stack machine

[18]We intend to show our students that the parsing problem is not a trivial one, and also that a general parser is too inefficient for a compiler.

8. a compiler for a language with arrays and structs, and, optionally, with procedures and functions which generates code similar to three-address code

The last two programs have to be written using `lex` and `yacc` (or `flex` and `bison`).

4.4 Evaluation of students' work

Continuous evaluation of work in the classroom is accomplished basically through self-evaluation and inter-regulation (students have immediate information about their work in the classroom during the discussions in common)[19]; also, an "open door" policy encourages them to discuss any classroom activity with the teacher during an extended schedule of office hours. Evaluation is completed through a couple of written tests in each semester, consisting in previously unseen open problematic situations, where the methodological approach to the solution is assessed in addition to the quality of the solution produced.

The evaluation of laboratory work is based on a weekly monitoring of each student's work in each one of the assignments, as well as on a carefully designed test set for each one of the programs they produce. When an objective is not accomplished at a given point of the course, there are more opportunities to reassess it, because most of the proposed work is incremental in nature.

The teacher maintains a detailed record card for each student with all of this information, as well as with subjective notes on the student's performance or learning circumstances, where available. The final grade will be based on a global assessment of the student's learning process and attitudes.

4.5 Results of the new model

The results we have obtained after applying the *learning as a research activity* pedagogical model for one year are very encouraging: our students (and ourselves) have learned more about compilers than students from previous years, and will more likely remember the basics of compiler design and this course for several years. Also, we have been gratefully surprised

[19] Periodical notebook monitoring would be a very desirable evaluation tool, but it is almost impracticable due to the size of student groups.

by the skills of our students in more occasions than we could have expected[20]. On the other side, we have really enjoyed teaching compiler design this way, and we find ourselves actively doing research in an area which is of high interest to us. We are planning to extend this model to subjects other than compiler design, and we also want to involve as many teachers from our department as possible in this teaching and learning model. We are also willing to help teachers from other universities to try this method in their compiler design course (or in any other course in computer science).

A brief list of the positive results we have obtained could be:

- Better understanding of complex concepts (like LR parsing);
- Higher motivation of students from this year with respect to previous years' students[21];
- Ability to face new situations and problems.

Some of the problems encountered include:

- Students' error messages are not adequate for an end user, because they are not used to auto-evaluate their own programs as final products; also, we did not expect them to fail in this subject, so we were not able to correct this problem on time during year 1994–1995.
- Very few students were able to write correct SDTSs (syntax-directed translation schemes). We think that this failure is caused by the inability of students to prove the correctness of their SDTSs[22].

[20] We did not expect our students to easily find by themselves a general bottom-up algorithm for parsing, or to design a grammar which reflects the associativity and precedence of expressions in Pascal or C, for example, but some of them did!

[21] Abboud [8] notes the importance of motivation and self-confidence (our italics):

> A principle role of education is to empower students by providing them with the tools for learning, thus building their confidence so that they can be *in charge of their own growth and development*. If the instructor can provide an environment where a student can experience learning as a process, then the student gains the confidence to learn and discover. The acquisition of information is insufficient, rather it is the desire to learn and to have the tools to be in charge of one's growth that is the ultimate role of education.

[22] During year 1995–1996 we are trying to correct this problem by forcing them to implement their compilers with `yacc` and `lex`, so they have to design an SDTS by hand before implementing the compiler.

In addition, a survey taken after the first semester of 1994 (70 students answered out of 220) gave very encouraging results:

- 89% of students thought that the new method used in the classroom is better or much better than classical teaching methods;
- 85% believed that active participation in classroom activity improves concept learning, whereas 89% believed that active participation helps understanding the teacher's explanations;
- 89% believed that working in groups is a good idea;
- 91% believed that they would be able to succesfully solve a real-life, open, compiler-related problem;
- When judging course quality in a scale from 0 to 10, Compilers I had an average overall mark of 7.1, compared to 4.7 for the rest of subjects they took.

An interesting aspect is that 66% of the students considered themselves 'good' or 'very good' at programming in C (in contrast with our findings: see section 3.3). We plan to take a thorough survey at the end of year 1995–1996 to reassess our methodology and practice.

5 Future developments and challenges

We are currently working towards three directions:

Redesigning the program of activities: Sometimes, an activity which is designed to lead the students to learn a new concept does not achieve its purpose and the students do not learn the concept or learn a misconception. In that case, the teacher has to redesign the activity *on the fly* and re-orient the discussion, or sometimes he or she has to drop the activity and improvise a new one. The whole process has to be done while in the classroom, and when the classroom session ends, the teacher has to rewrite that activity or block of activities for the next year. The process may never converge, due to changing conditions such as the students' background, the teacher, or changes in previous activities.

Team-oriented work: While in classroom sessions our students work in small groups, laboratory work has to be presented individually, although the students can still work in groups. We also think that the laboratory assignments may be too laborious for a single student, so we are considering letting students complete the assignments in small groups (the same groups formed for classroom sessions). One important reason to do this is that team-oriented work is critical to problem solving[23]. The assignments would have to be redesigned, and we would be much more demanding in some aspects —more related to software enginering than to compiler design[24]— like specification, design, and documentation of each program. A simplistic application of this team-oriented approach might have a negative effect: lazy students or (which is worse) less brilliant students may remain *hidden* into the groups and they could even pass the course without having learned the basics of compiler design. A solution to this problem would be an individual final exam, but we think that this solution would not motivate these less prepared students to *learn*, it would only prevent them from passing the course. Our current approach (team work in classroom sessions and individual work in laboratory sessions) allows us to guide very closely each student's work, which also gives us a fair subjective impression of the student's understanding of the basics of compiler design. This new approach should include this individual feedback between the student and the teacher, which is of high importance in our model (see section 2.5).

Project-oriented course: Another improvement would consist in proposing our students a large project at the beginning of the course (which could be the design and implementation of a compiler for a complete high-level language like Pascal or C, which would have to be carried out by several groups of aproximately four students) and redesign the program of activities

[23]This aspect is also emphasized by other researchers, like Abboud [8] (our italics):

> Activities that are useful and promote learning are group endeavors. Group projects, such as working jointly on a problem, provide an environment to learn collaboration and skills that go beyond the subject matter. Students perceive the need to communicate and start acquiring that skill. They also *learn to consider alternative approaches to problem solving, and to reflect on them and evaluate them.* Moreover, group dynamics provide a gain in self-knowledge and a way to develop self-confidence.

[24]As stated in section 3.3, our students have a very shallow background on software engineering (not only due to the organization of their curriculum), and we think we should also make emphasis on these aspects while teaching compiler design, because building a compiler is a good opportunity to practice software engineering techniques.

to follow the project as closely as possible, that is, we should extract the common problems of all the languages for which we would have proposed compiler projects and construct a set of activities for them (in this part of the course the groups could work in parallel); then we should design specific blocks of activities for each special construct of each language (like pointers, register types, procedures and functions, etc.) and give one of these blocks to each of the groups that is assigned to each compiler project. We could even let the students design their own high-level language. We think this project-oriented course would achieve the goal of teaching all students the basic techniques and algorithms needed for compiler design, and also would develop the skills of the students for working with a number of other groups, which involves previous specification and documentation of the modules they are going to implement and also involves developing communication skills, as recommended by the ACM/IEEE-CS Task Force report [9]. There would be coordination meetings [18] between all small groups assigned to each project.

The project-oriented approach would be very challenging due to the large amount of students that attend our courses (aproximately 200 students per year). We intend to initially try this project-oriented approach with a small fraction of all students.

Acknowledgements: The authors wish to thank Joaquín Martínez-Torregrosa, Albert Gras-Martí, Rafael C. Carrasco, Eva Gómez-Ballester, Domingo Gallardo and A.M. Corbí for their inspiration and for their careful reading of this manuscript.

References

[1] D. Gil-Pérez and J. Carrascosa-Alis, "Science learning as a conceptual and methodological change", *European Journal of Science Education*, Vol. 7, No. 3, 1985, pp. 231–236.

[2] D. Gil-Pérez and J. Carrascosa-Alis, "Bringing Pupils' Learning Closer to a Scientific Construction of Knowledge: A Permanent Feature in Innovations in Science Teaching", *Science Education*, Vol. 78, No. 3, 1994, pp. 301–315.

[3] D. Gil-Pérez and J. Martínez-Torregrosa, "A model for problem-solving in accordance with scientific methodology", *European Journal of Science Education*, Vol. 5, No. 4, 1983, pp. 447–455.

[4] R.M. Garrett, D. Satterly, D. Gil-Pérez and J. Martínez-Torregrosa, "Turning exercises into problems: An experimental study with teachers in training", *International Journal of Science Education*, Vol. 12, No. 1, 1990, pp. 1–12.

[5] D. Gil-Pérez, A. Dumas-Carré, M. Caillot and J. Martínez-Torregrosa, "Paper and pencil problem solving in the physical sciences as a research activity", *Studies in Science Education*, Vol. 18, 1990, pp. 137–151.

[6] D. Gil, J. Carrascosa, C. Furió and J. Martínez-Torregrosa, J., *La enseñanza de las ciencias en la educación secundaria*, Col. Cuadernos de Educación, ICE-Horsori, Barcelona, 1991.

[7] J.D. Novak, "Human constructivism: towards a unity of psychological and epistemological meaning making", *Second International Seminar on Misconceptions and Educational Strategies in Science and Mathematics Education.* Cornell University, Ithaca, New York, 1987.

[8] M.C. Abboud, "Problem Solving and Program Design: A Pedagogical Approach", *Computer Science Education*, Vol. 5, No. 1, 1994, pp. 63–83.

[9] A.B. Tucker, B.H. Barnes, R.M. Aiken, K. Barker, K.B. Bruce, J.T. Cain, S.E. Conry, G.L. Engel, R.G. Epstein, D.K. Lidtke, M.C. Mulder, J.B. Rogers, E.H. Spafford and A.J. Turner, "Computing Curricula 1991: Report of the ACM/IEEE-CS Joint Curriculum Task Force", ACM / IEEE Computer Society Press, 1990.

[10] A.B. Tucker, B.H. Barnes, R.M. Aiken, K. Barker, K.B. Bruce, J.T. Cain, S.E. Conry, G.L. Engel, R.G. Epstein, D.K. Lidtke, M.C. Mulder, J.B. Rogers, E.H. Spafford and A.J. Turner, "Computing Curricula 1991: A summary of the ACM/IEEE-CS Joint Curriculum Task Force Report", *Communications of the ACM*, Vol. 34, No. 6, June 1991, pp. 68–84.

[11] J. Clement, D.E. Brown, and A. Zietsman, "Not all preconceptions are misconceptions: finding 'anchoring conceptions' for grounding instructions on students' intuitions", *International Journal of Science Education*, Vol. 11, pp. 554–565.

[12] J.P. Tremblay and P.G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill, New York, N.Y., 1985.

[13] J. Elder, *Compiler Construction: A Recursive Descent Model*, Prentice Hall, Hemel Hempstead, England, 1994.

[14] B. Teufel, S. Schmidt and T. Teufel, *C^2 Compiler Concepts*, Springer-Verlag, Vienna, Austria, 1993.

[15] R. Wilheim and D. Maurer, *Compiler Design*, Addison-Wesley, Reading, Massachussets, 1995.

[16] C.N. Fischer and R.J. LeBlanc, Jr., *Crafting a Compiler with C*, Benjamin/Cummings, Menlo Park, California, 1991.

[17] A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, Reading, Massachusetts, 1986.

[18] Rafael C. Carrasco, personal communication, 1995.

[19] D.A. Watt, *Programming Language Processors*, Prentice Hall, Hemel Hempstead, England, 1993.

A Course outline

In this section we give an outline of the program of activities for year 1995–1996. Although it covers basically the same subjects we covered in the program for year 1994–1995, we have rewritten most of the activities and we have reordered some blocks both to address the problems we had and to explore and correct some misconceptions detected during year 1994–1995. As we explained in section 5, this outline is open to future reorganization, in order to improve the learning process. The relation between each block of activities and the laboratory assignments listed in section 4.3 is also noted.

1. *What are we going to study?:* This block introduces the students to the pedagogical model, tries to motivate them, explores their previous ideas about compilers, and begins a study of what a compiler's task is. Laboratory assignment #1 starts while studying this block.

2. *The construction of correct programs:* Understanding the concept of programming language syntax and learning ways of specifying it are two important aspects covered in this block. Also, this block deals with the concept of *a program*'s syntax and with ways it can be represented (trees, parses). This block also invites the students to outline an algorithm to obtain a program's syntax from the program itself. By the end of this block our students will start to work on laboratory assignment #2 (a general parser for context-free grammars).

3. *Something like a compiler:* This is a very important block. It has a double purpose: to make our students learn that it is not possible to translate a program without knowing its syntax (structure), and to challenge a possible misconception: that compilers first check syntax, and then generate code (see section 4.1.4). In this block, students outline the workings of a small compiler (see appendix B, activity 23).

4. *A program is more than a character string:* This block basically covers the phase of lexical analysis. The main objectives of this block are: to make our students aware of the convenience of processing the program at a higher level than that of characters, and to learn how to write the lexical specification of a language and how to implement it both as program code and as transition diagrams.

5. *Top-down analysis:* This block is mainly dedicated to recursive-descent parsing. It also covers subjects like transforming a grammar to make it suitable for top-down parsing, detecting syntax errors as soon as possible, and writing good error messages. At the end of this block students start to work on laboratory assignment #3, a recursive descent parser.

6. *Write a whole new parser for each new language?:* This block covers table-driven LL(1) parsing. It also makes emphasis on the connection between LL(1) parsers and

pushdown automata, and, as the previous block, this one covers the adequate detection of syntax errors and the construction of good error messages. By the end of this block our students start laboratory assignment #4, a LL(1) parser.

7. *Bottom-up analysis:* This block deals first with bottom-up parsing in general, and with shift-reduce parsing in particular, and then covers the construction of SLR(1) tables. It also deals with error detection and error messages. As in previous blocks, when this one will be finished, our students start laboratory assignment #5, a SLR(1) parser.

8. *What are we going to study (II[25])?:* After some blocks dealing with parsing, this block recalls the subject of syntax-directed translation, and serves as an introduction to semantic analysis and code generation.

9. *How can we use syntax to translate?:* This block has two main objectives for our students: to learn the basic techniques of syntax-directed translation, and to learn some formalisms for specifying a translation, such as syntax-directed translation schemes (SDTS) and attribute grammars. The implementation of SDTS on top-down and bottom-up parsers (including `yacc`) is discussed. Students start to work on laboratory assignment #6 (a small toy compiler) while working on this block.

10. *Got to know these types before working with them:* This block covers aspects related to data types and their compilation, and introduces the notion of type system. The construction of SDTS for type checking, the concepts of type equivalence and type compatibility are also covered. Laboratory assignment #7 requires our students to implement a type system in a compiler for a Pascal-like language, and that is why the assignment starts when this block ends.

11. *A thousand languages, a thousand machines ..., a million compilers?:* This block is mainly dedicated to the study of intermediate representations: abstract stack machine code, syntax-based representations and three-address code. Our students will have to write a toy compiler generating three-address code (laboratory assignment #8) by the end of this block.

[25] Due to the scheduling of the compiler design subject in two separate semesters, Compilers I and Compilers II, this block is necessary as an introduction to the second subject.

12. *Object code as good as hand-made?:* In this block, students are introduced to object code generation and optimization.

B A sample of classroom material

This section contains some activities extracted from the program we follow in our course. The first two have been taken from block 1, the third from block 2, and the fourth from block 3. Each activity is followed by a short explanation of its purpose, that is, which concepts we want our students to learn.

> 5. Put together all you know about compilers, and you will see that it is more than you would have thought. What does a compiler exactly do? Why do programmers need compilers? Why don't we write our programs in assembler? Think about how you would explain it to someone who has only a basic knowledge about computers. Think also about the compilers you have worked with during past years.

This activity searches into the concept of *compiler* that our students have, and forces them to make a definition of what a compiler is (that is, listing the tasks performed by a compiler). After the discussion in common, it must be clear that compiling means translating from one source language to an object language on a given platform, and that a compiler is a programming-language translator. The discussion should also deal with the advantages of high-level languages (data types, type checking, flow-control instructions, etc.) versus low-level languages like assembly language.

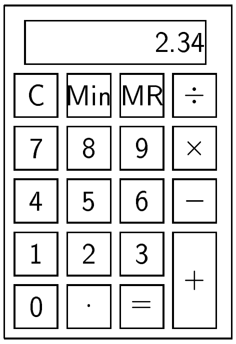
> 6. How would you completely specify a given compiler?. Think about compilers you already know to make your specification more complete and accurate.

This activity serves to study in more depth the concept of compiler developed in the previous activity. The specification schemes proposed by the groups will be discussed in common and criticized by other groups. Every specification proposed should include the basic elements (source language, object language, the compiler's host operating system, the operating system where the object code will be executed, etc.).

20. How could you trace back how a given program has been built, starting from the program itself and from the building rules (syntax) of the source language? How is this operation usually named? Could you outline a general method to perform it? Try it with a sequence of commas and letters to see if it is a *sequence of letters*. When it is not, can we know why?

We intend to show parsing as a method to reveal how a program has been built. Of course, an important secondary effect of syntax-analysis should be to know if it is possible to build the program from the syntax of the language, that is, to know whether the program belongs to the language or not. In the following activities we should show that it is more important to obtain a representation of a program's syntax than to simply say if it is correct or not.

23. Imagine you are given an arithmetic expression (with integer and fixed point numbers, additions, substractions, multiplications and divisions, and parentheses, with the syntax, precedence and associativity of these operators in Pascal or C) and you are told to evaluate the expression using this calculator:



The keys $\boxed{\text{Min}}$ and $\boxed{\text{MR}}$ are used to access ten memories numbered 0 to 9. Thus, the sequence $\boxed{\text{Min}}\,\boxed{9}$ stores the value in the display into memory 9, and the sequence $\boxed{\text{MR}}\,\boxed{5}$ brings the value stored in memory 5 to the display. In all other aspects, this calculator operates like a regular model. What sequences of keypresses would you use to evaluate the following expressions?

- `3+4*5`
- `3-4*(4+1)`
- `1+2+3+4+5-6+7-8-9-10+12.45`
- `10/(4-(2*0.11/0.1))`
- `(3+4)*5-4-3`
- `10.3*(0.1+9.11*(2/0.05/3-8.25))`

> Try to design an algorithm (first try to act like an algorithm) that generates the sequences of keypresses in a systematic way, reading the expression from left to right and considering that the code generated must be correct for the calculator. Show in detail which rules you have used to decide which part of the expression must be evaluated before other parts, and why. How have you managed the ten memories?.

Before dealing with the compilation (translation) of a language, we can think about its interpretation. In the case of expressions, this involves evaluating them before considering how to translate them to a sequence of keypresses. In this activity we intend to show that simply reading an expression from left to right is not enough to evaluate it on a pocket calculator, and therefore our students have to study another approaches to the problem.