Lab assessments in undergraduate course in Compilers for students with no prior knowledge in assembly

Conference Paper · May 2014

DOI: 10.1109/MIPRO.2014.6859663

4 authors:

Vesna Kirandziska
Ss. Cyril and Methodius University in Skopje
28 PUBLICATIONS **120** CITATIONS
SEE PROFILE

Mile Jovanov
Ss. Cyril and Methodius University in Skopje
63 PUBLICATIONS **158** CITATIONS
SEE PROFILE

Marija Mihova
Ss. Cyril and Methodius University in Skopje
66 PUBLICATIONS **164** CITATIONS
SEE PROFILE

Marjan Gusev
Ss. Cyril and Methodius University in Skopje
487 PUBLICATIONS **2,081** CITATIONS
SEE PROFILE

Lab assessments in undergraduate course in Compilers for students with no prior knowledge in assembly

Vesna Kirandziska, Mile Jovanov, Marija Mihova, Marjan Gusev
Faculty of Computer Science and Engineering
Ss. "Cyril and Methodious University" in Skopje
{vesna.kirandziska, mile.jovanov, marija.mihova, marjan.gushev }@finki.ukim.mk

***Abstract*—Compilers is an important course in the curricula of Computer Science Faculties. A problem that we have faced while organizing the course at our institution was lack of prior knowledge of some students in the area of assembly programming and microprocessors. Due to the fact that the number of classes for the course was limited, we were not able to teach the necessary material on assembly programming. In this paper we present our idea for the lab project to be done by students through the course, that should enable them to grasp all necessary concepts in Compilers without a need of assembly programming. The idea was tested a few of years in a row, and most of the students managed to finish the project, and achieved better results in the final exams.**

***Index Terms*—compilers, learning**

I. Introduction and related approaches

By definition, a compiler is a computer program by which a high-level programming language is converted into low-level programming language that can be acted upon by a computer. Hence, for someone to be able to make a compiler, she needs to know the high-level programming language, and also to have a great understanding of the low-level language.

The course Compilers or similar courses named Compiler construction, Compiler and interpreter, Compiler Practicum, Programming languages and Translators are all in the computer science program at many Universities. At our Faculty of Computer Science and Engineering in Skopje this course is elective course in the sixth semester, by the proposed program. It is a compulsory course for students enrolled in some of the modules like 'Computer science' and 'Software engineering'. Each year we have around 40-70 students enrolled in this course. It is of a significant importance how this course is taught to the students and whether they can achieve the course goals.

There are different experiences worldwide. At the University of Virginia [1] a big accent in the curriculum is given to code generation. The course project consists of two programming assignment. The first is building a compiler for COOL (the classroom object-oriented language which is a high-level programming language) and the second is optimizing it. COOL has the essential features of a realistic programming language, but is small and simple enough so that its compiler can be implemented in a few thousand lines of code. The course Compilers [2] held at Stanford University also uses the COOL language as an example for building parts of a compiler through five different programming assignments. Even more, an optional course project is to write a complete compiler for COOL in C++ or Java.

Other universities use part of Java programming language as the high level-language that has to be translated. For example, at the University of Texas[3] as a student assignment, each student writes a compiler for Pascal and code is generated for a real processor and run on hardware. Beside heavy programming skills, hardware knowledge is crucial for creating a program written in a low-level language. One of the four distinct assignments in the project at this university involves code generation. A major part of the course held at the University of Waterloo [4] consists of an implementation of a compiler for a simplified Java-like language called Joos1W. The course project represents 75% of the student grade.

At all mentioned universities the project assignment represents the great deal of the student evaluation. Building their own compiler enables students to get more deeply involved in the crucial parts of a compiler and it enables understanding it better through all challenges that came up while working on their programming assignment.

Other thing in common for all Compiler courses lectured at different universities is that the compiler built usually translates in some version of an assembly programming language. At Stanford [2] the MIPS (Microprocessor without Interlocked Pipeline Stages) assembly is used, while at the University of Waterloo[4] use the Netwide Assembler dialect that runs on he Intel x86 architecture. So, for building a translator into low-level language the knowledge of assembly language is essential.

All programs for Compilers at other universities discussed above require knowledge form Data Structures, Algorithms, Programming, Microprocessors and Formal Languages.

At the Faculty of Computer Science and Engineering in Skopje Structural and Object Oriented Programming are courses taught in the first student year (by the proposed schedule), Data structures, Algorithms and Formal Languages are in the second student year, but the course Microprocessors is an elective course in the third or forth student year. The

curriculum of Microprocessors includes assembly language and programming 8086 microprocessor. As a result, usually students enrolled in the Compiler course, had not been enrolled on the course Microprocessors previously. This was a problem that had to be considered.

In this paper we are going to present our solution on how to teach students Compiler with no previous microprocessor and more precisely assembly knowledge.

In the next sections our idea for organizing the Compilers classes as well as the rules for student evaluation are presented. Afterward, our approach for presenting code generation to our students is explained in more details. Next, the results taken from our four year experience in teaching Compilers are presented. At the end a conclusion is given.

II. Course curriculum

The topics of this course follows the book "Compiler Construction: A Practical Approach" [5] and its main objectives are students to get familiar with a compiler and also to learn how a compiler is build. Here the Inger language is defined and used as an example language for which the compiler is built.

'Compilers' weights 6 ECTS credits and is organized with 5 classes a week: 2 classes for lectures, 1 for auditorium exercises and 2 for laboratory exercises. The lectures continuously cover each part of a compiler construction. In the curriculum of the Compiler course all parts of a compiler are considered in details.

The language for which a compiler is made can be represented through syntax diagrams, Backus-Naur Form or Extended Backus-Naur form. These approaches for language representation are contained in the first lectures.

Following these lectures lexical analysis is introduced to students. The part of the compiler which perform lexical analysis consists of a lexer implemented as a finite automaton and a parser implemented as a push-down automaton. Beside checking if one program is lexically correctly written, a compiler could detect and report the type of error that is found in a program and even where the error is found. This is also included in the curriculum.

Next, the part of the compiler concerned with checking the semantics of the programed code is presented. Between many different kinds of semantic errors that can be found, most of them involve variables or symbol names. A symbol table is usually used as a tool in semantic analysis and is included in the course.

The last part of the lectures is about the code generation module of a compiler. This is actually what we are concerned about in this paper. The classes dedicated to this part of the compiler are given in two lectures. The dedicated time does not allow the professor to teach about this part of a compiler, and additionally to show the basics of an assembly language since our students usually do not have any background knowledge in microprocessors. Consequently, we used our approach presented in this paper to teach students about this part of a compiler.

While during the lectures we cover the topics and we use auditorium exercises for more practical examples, at the laboratory exercises students work on the practical lab assignments given. In the next subsection the student evaluation for this course is presented.

A. Students' grading

Students are graded by the midterm exam(20%), final exam (25%), homework(15%) and by the grade for the final project assignment (40%). Exams cover the theoretical knowledge in the Compiler construction. Homework is different in different years, and for 2 years we have included an original on-line collaborative activity named "Ontology" also developed at our insitution [6].

Here we present the work on the project which was constructing a compiler. The whole project consists of several separated tasks:

- **Task 1:** Make a syntax diagram for the language A
- **Task 2:** Program a lexical analyzer (lexer) for the programming language A
- **Task 3:** Program a parser for the programming language A
- **Task 4:** Make a complete syntax analyzer with error detection
- **Task 5:** Make a complete semantics analyzer with error detection
- **Task 6:** Generate code in language B from the program in language A
- **Task 7:** Test the compiler

Divided in teams with 3-5 students in each group, students should do the assigned tasks needed to do the final project. Their involvement in the project should be on weekly basis, by upgrading the compiler with the newly learned module introduced on the lectures. The laboratory exercises are consultation exercises in which students ask question about their project and advance to its construction on regular bases.

Working in teams allows some students to work more then others and be more involved in the project, while others can be not involved at all. This arises the problem of how to grade students. One solution was giving the students a chance to grade by them selves their activity in the project. After grading the project itself, the students should all get the appropriate part of the points based on their involvement. Another approach is the teacher to grade each student in the group from an oral presentation about his/hers involvement in the project. Both approaches were used in the Compiler course in different years.

Because different students have different possibilities, background knowledge and ambitions, several variations of the final project were given. The hardest project scored 100 points, while the easiest one 65 points. In this way the projects ware leveled to the student knowledge. Students were given a possibility to choose the project level they think that suits them. Afterward, teams were formed out of the survey results.

In the next section the project assignment is defined in more details.

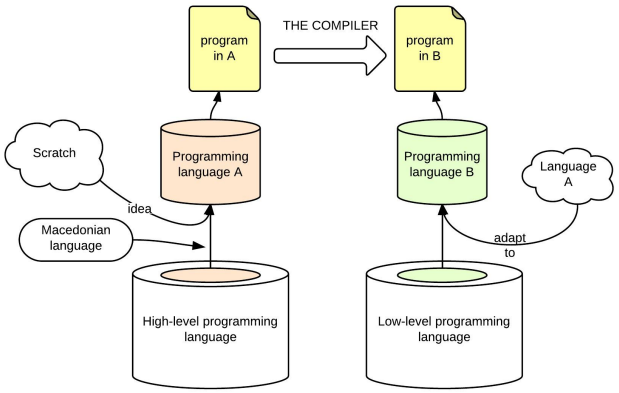

Fig. 1. A schematic view of languages A and B

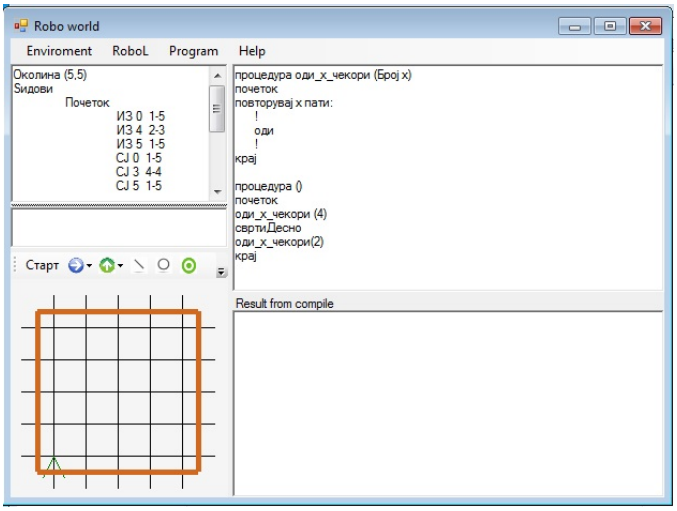

Fig. 2. A shell application for the compiler

B. Project Assignment

The project was stated as follows: Make a compiler for one given programming language A and translate it to another assembly-like programming language B.

The languages A an B are new languages that were introduced in our course. They are created just for this specific purpose. In all years the course was held, new (similar but different) languages A and B were introduced. On 1 a schematic view of these languages compared to existing programming languages is shown. As shown on the figure the programming language A is based on the procedural high-level programming languages like C, Basic or Pascal. A is not an object-oriented language. It includes the three most basic statements and other basic features. The key words in the language A are written in Macedonian language and so it differs form all other languages.

The inspiration for this language comes from turtle-based programming languages used for beginners in programming languages. Indeed, the elementary statements of the languages are actually commands for the movement of the specific turtle-object like a frog, robot, bird and so on. We have used all these objects in our course. The interesting content of the language and its statements made the project more interesting to students. As an example, a robot's program should move him in a specified environment and it must be careful not to hit the walls. Another example is a frog's program. This program should enable a frog to jump form one field to another considering water in the sense it should not step on it. Using these languages as introductory languages for young students that show interest in informatics, gives great results and we have included them in the strategy for education in informatics in our country [7].

As stated, these programs should be executed in a specific environment. A language for environment definition was used to define the environment. But, program execution is not in the scope of the course Compilers because here an interpreter of the language is needed.

The programming language B as explained on Fig. 1 is based on assembly, but has only the most basic statements needed for the translation to be possible. All hardware knowledge for the execution of a program written in this language is excluded.

In the rest of the paper we are going to explain in details only one of the project assignments (given in 2009). The language A defined was called RoboL (extension - rl) and was specialized for moving a robot in its environment. For the project assignment the students were given a shell application. It is shown on Fig. 2. This is just the interface of the compiler students should do. In this application a compiler for the language should be build in. The application is build in Microsoft Visual Studio. In the lower left corner a picture for the robot environment is given. Bold lines represent walls. On the right side a program written in RoboL is shown and on the left side a program for environment definition is given. The menu of this application should be implemented by students.

In the example project RoboL was translated into RIMAL (Robo Imaginary Meta Assembly Language). More about these languages will be explained in the next sections. The result from the compilation is the program translated in the RIMAL language (extension - rml) if the program is correct. But if the program is not correct, students should output messages for all errors found in the program code. These messages should be displayed in the lower right part of the application window.

III. The Languages for the Project

Next, the languages used for building a compiler are introduced.

A. The high-level language

RoboL poses the three simple, hierarchical program flow structures: sequence, selection, and repetition. There are 3 basic statements which are direct commands for the robot movement: go straight (one step), turn left or turn right.

```
процедура ()
почеток
променлива к,бр
к=4
бр=0
ако ознака
!
  бр=1
!
повторувај к пати:
!
    оди
    ако ознака
    !
      бр=бр+1
    !
!
                          example.rl
```

```
свртиДесно
повторувај до (бр==0):
!
    оди
    ако ознака
    !
      бр=бр-1
    !
!
крај
```

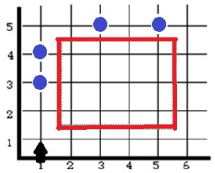

Fig. 3. An example program in RoboL and an environment in which it should be executed

Selection structure is realized trough the if-like statements. While-like repetition is not implemented, but a variant of do-while statements and for-statements is present in the language. There are five different variants of repetition (loop) statements.

In the language only integer and a kind of character types can be declared. The character variables are specific variables that can be considered as an enumeration. Indeed, these variables contain information about the direction of the robot position that can be South-S, North-N, West-W and East-E. As a result, arithmetic operators have no meaning with these character variables. Integer variables can be only positive. Arithmetic operators for addition, subtraction and multiplication can be used with integer variables and constant values. For logical expressions relational operators for comparison are used. These can be used as conditions in selection and repetition statements.

Other conditions used in these statements address the environment in which the robot is. The robot environment has walls (red lines) and marks (blue lines). In each moment a robot is in one point that is an intersection of one vertical and one horizontal line. Also the robot heads in one of the four directions (S,N,W,E). Considering this, other conditions implemented in the language are based on:

- **Wall**: If the robot is exactly in front of a wall the expression will be true
- **Mark**: If the robot is on a mark the expression will be true.
- **Some direction (S,N,W or E)**: If the robot is pointed in the stated direction the expression will be true

RoboL supports the concept for procedures. Functions and return statements are not supported.

The definition of the language was given to the students with a written tutorial. Example programs in this language were also available.

B. The low-level language

As stated in the Introduction, because our students in general have no knowledge in assembly as low-level language we made an artificial language RIMAL simulating a real assembly.

Robo Imaginary Meta Assembly Language (RIMAL) is a simulation of a simple assembly. The idea of this language is not to be used for program execution but to be used for educating students. The code written in this assembly-like language can not be executed by some hardware. This language has the most basic features of an assembly. Indeed, the language has similar instructions as the assembly language. But, some of the instructions are omitted from this language and some are added to it. The final product of student project beside giving error messages for the syntax and semantic errors, is the code generated in RIMAL. By making this code students learn and practice how code generation is implemented in a compiler.

As in assembly 8086, in RIMAL registers, variables and stack can be used for storing data. Because in the high-level language RoboL arrays are not implemented other memory are not used in RIMAL. 3 registers can be used:

- regN - for storing integer data
- regD - for storing character data
- regC - for constant values

As shown, all registers have a specific usage and should be used for the appropriate data. Note, that although logically there are registers in the language, there are no actual registers. Other important similarity to assembly language is the usage of labels to label code lines. In RIMAL also, any string different then the key words can be labels. After the label : (colon) should be written.

Instruction type	Instructions	Assembly
Move instructions	move	modified
Arithmetic instructions	inc add sub mul	✓
Function instructions	call, ret	✗
Comparator instructions	cmp	✓
Robot movement instructions	rl (go left) rr (go right) go (go)	✗
Jump instructions	je, jne jl, jle jm, jme	✓
Stack instructions	push, pop	modified

TABLE I
RIMAL INSTRUCTION SET

In Table I the instruction set of the newly defined language is represented. For each instruction type the instructions are given. Also, a comparison with the assembly language is roughly given. For example, move instructions which are used for coping data from one location to another is modified with respect to the mov instruction in assembly. Beside the instruction name, the operands can only be: variables and registers. Similarly, the stack instructions are modified so that the instruction 'push' pushes the value in the register regN on the stack.

Arithmetic, comparison and jump instructions are similar with the instructions in assembly language. There is a difference only in the operand values in these instructions.

```
odi_do_x_oznaki:
data x
mov regN,x
start:
go
cmp reC,$M
jne next
sub regN, 1
next:
cmp regN,0
jne
ret
main:
data k
data br
move k, 4
move br, 0
cmp reC,$M
jne next
move br, 1
next:
```

```
move regN,0
start:
go
cmp reC,$M
jne next
push
move regN, br
add regN, 1
move br, regN
pop
next:
cmp regN,k
jne
rr
call
odi_do_x_oznaki(br)
ret
```

Fig. 4. An example program translated in RIMAL

Call instructions are different then in assembly language. The call instruction has one operand which is the label where the function is written in the code. But in assembly the operand is the address where the first instruction in the function is written. When calling a function in assembly the IP (Instruction pointer) register is pushed on the stack and the address of the new instruction is put in the IP. But, here the knowledge for the instruction pointer and also the instruction stack is omitted. In in RIMAL the function definition ends with the key word ret. After the code in the function is executed the program continues after the function call. This approach allows does not allow recursion functions.

The RIMAL language is enriched with three robot movement instructions. Because in the RoboL language there are instructions for moving the robot straight ahead, left or right by one step, RIMAL also has some version of these instructions. These are the instructions go, left and right, accordingly.

As a conclusion RIMAL has a very similar syntax with assembly. The valuable feature of RIMAL is that students are not concerned with the semantic of its instructions and their execution. This help students to see this language just like another (non high-level) programming language. Computer hardware on which the low-level language should be executed is not known. This makes the problem of understanding the language easier, but also gives students a little sense of what an assembly language looks like.

In Fig. 4 an example code translated in RIMAL is given.

IV. Results from the lab project assignment in the Compiler course

With the laboratory project given, successful student project included more then 5000 lines of programming code for building a compiler. By doing the project students went through the whole process of making a compiler by which they practically learned what a compiler is and how does it work.

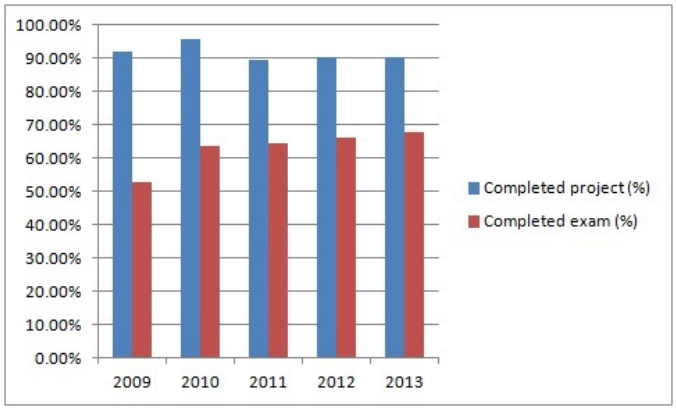

Fig. 5. The percentage of students with completed projects compared to the percentage of students with completed exam

Students worked in groups with 2-5 students in each group. In the laboratory exercises all team members could consult with their lab demonstrator or teacher about the project. In each exercise students were given a specific task towards creating the compiler which was the end objective in their project. As a result the compiler was built step-by-step on weekly bases. Each class students were given a chance to ask some questions about the task from the previous week. This approach gave students many opportunities to learn more about their project and to improve on their teamwork skills.

Also this approach yield great results in completion of the student projects. On Fig. 5 a graph for the percentage of students that completed the project is given. The results presented are since 2009 until today. Only around 10% of the students do not complete the project at the end every year. In 2010 only 4% of the students did not make a compiler enough for passing the exam. Even more in 2012 15 out of 17 given projects, which is more then 88%, were completed.

The percentage of completed projects is a promising result. There may be several reasons and some of them are given here.

First, working in groups is a challenging, but motivating task in which students make a great effort to do the assignments given. Groups may have different profiles of students and due to their difference they could not only get better results, but also learn from their teammates.

Another explanation is that students choose their project from more project variants with different difficulty levels. Because students are usually aware of their skills, knowledge and ambitions they choose a project with appropriate difficulty and in most cases manage to complete the project. As a result, suitable projects are available for each student. The most difficult project that can be chosen includes visualization module with which a program in RoboL is interpreted. This is the most interesting project for students, and each year more and more students are interested in taking the most difficult,

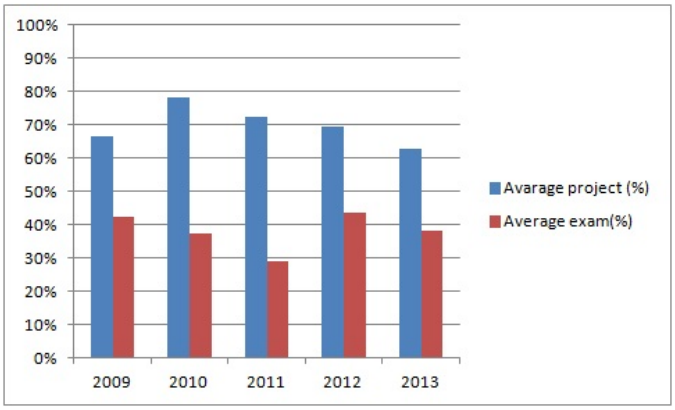

Fig. 6. Comparison of student project assignment and student exam percentage in 2009-2013

but most interesting project.

The last explanation for the high percentage of project completion is due to the organization of the laboratory exercise. Step-by-step tasks given to students enable students to complete most part of the projects during the classes. The help from demonstrators on the laboratory exercises also improves the project accomplishment result.

In Fig. 5 the percentage of students that completed their project is compared to the percentage of students that have completed or passed the exam. Because the project covers all the material from the lectures it is expected that the project would improve the percentage of students who completed the exam. Every year, more than 50%, and up to 70% of the students were able to pass the exam.

In all years from 2009 to 2013 the percentage of students that completed the project is higher then those who completed the exam. This is expected because the project assignment is much more attractive and interesting because of, among other previously stated features of the project, teamwork included.

The project in general helps students learn the course material and pass the exam. Indeed, through students activity in the project, students are stimulated to learn the theoretical part of the exam, even though it is hard and rather ”boring” for most students.

The final course grade depends on the project assignment grade as well as from the final exam. Fig. 6 shows that students got higher results on the project then on the exam. The exam validates student's theoretical knowledge for compilers, while the project checks the practical understandings of compilers and its implementation in some programming language. Students could choose from Java and C++ as programming languages for implementation.

The average project grade given in percentage in the last 2013 was something above 60%. Best results are noticed in 2010 where the average student score was around 80%. The average exam grade in 2010 was little below 40%.

The difference between the the project and exam grade (%) is notable decreasing from 2011 until today. In the future more close average grades for the project and exam are expected. Students should be motivated to work more on the project. One possibility is to assign roles for each student. Also, a group manager could be assigned. The manager should give an evaluation for the effort and activities of each student in his/hers group. The fact stated that working on the project helps students learn the theoretical material is our assurance for future improvement.

The last accomplishment of our approach presented here is that most part of the students managed to learn the RIMAL language without previous knowledge of microprocessors. Truly, completing a project means that students also did the code generator part of the compiler where RIMAL language was used for translation. Even though they did not have any knowledge in assembly, they could understand this language and use it in the project.

V. Conclusion

In this paper we discussed the organization of the course Compilers at our Faculty of Computer Science and Engineering. We presented our original method of teaching compiler production trough a practical project in the conditions where students does not necessary have prior knowledge of an assembly language and in the field of microprocessors.

The presented results from 5 years teaching the course confirm that this approach is efficient, and it can be used by others when similar conditions occur. This was the main intention of the authors when writing the paper. We plan to continue our work in the following years, making adjustments with every following generation of students.

Acknowledgement

The research presented in this paper is partly supported by the Faculty of Computer Science and Engineering in Skopje.

References

[1] “The university of virginia engineering: Compilers practicum,” 2014. [Online]. Available: http://www.cs.virginia.edu/

[2] A. Aiken, “Compilers,” 2014. [Online]. Available: https://www.coursera.org/course/compilers

[3] G. S. NovakJr., “Compiler construction,” 2014. [Online]. Available: http://www.cs.utexas.edu/~novak/cs375.html

[4] “University of waterloo: Compilers,” 2014. [Online]. Available: http://uwaterloo.ca/

[5] F. Benders, J.-W. Haaringm, T. Janssen, D. Meert, and A. van Oostenrijk, *Compiler Construction: A Practical Approach*, 2003.

[6] M. Jovanov, M. Gusev, and D. Martinovikj, “A new model of on-line collaborative activity for building ontology in e-learning,” in *Proc. of the IEEE Region 8 Conference EUROCON 2013)*, 2013, pp. 25–31.

[7] M. Jovanov, B. Kostadinov, E. Stankov, M. Mihova, and M. Gusev, “State competitions in informatics and the supporting online learning and contest management system with collaboration and personalization features mendo.” *Olympiads in Informatics*, vol. 7, 2013.