# Learning Compiler Construction by Examples

**José de Oliveira Guimarães**
Departamento de Computação – UFSCar
São Carlos – SP, 13565-905
Brazil
email: jose@dc.ufscar.br

Abstract: In traditional compiler construction courses, each compiler phase is studied in detail before moving on to the next one. This not only places a great distance between theory and practice but also makes the students lose the big picture of the subject. We have been employing a different course format in which the subject is incrementally introduced through ten compilers of increasingly complexity. The first compiler is in fact just a syntax analyzer of a very simple language. The last one is a complete compiler of a Pascal-like language. Students of this course learn how to build compilers faster than the usual.

Keywords: compiler construction, learning by examples, object-oriented programming.

D.3.4 [Programming Languages]: Processors - Code generation, Compilers, Parsing

## 1. Introduction

Students usually consider difficult courses on compiler construction. The reasons are that compiler construction demands a heavy dose of programming and theory. A compiler operates in phases, each one with its particularities, algorithms, techniques, and tricks of the trade. The phases are lexical analysis, syntax analysis (parsing), semantic analysis, code generation, and code optimization [5]. The last phase is not usually studied in the undergraduate courses of our university.

A compiler takes a program written in a source language S and produces as output another program in a target language. The lexical analyzer takes characters of the input, in language S, and groups them in what we call tokens. Each language terminal is a token, which is associated to an integer constant. The syntax analyzer (parser) takes the tokens as input and checks if the source program matches the S grammar. The parser may build an abstract syntax tree (AST) of the source program. An AST is a data structure representing all the main elements of the input. It has all the important information present in the source program. The source program may have errors not detected by the parser such as "variable not declared" and "left and right-hand sides of the assignment have incompatible types". The semantic analysis is responsible for this kind of checking. In general, the semantic analyzer is composed by a myriad of pieces of code spread in the parser. The code optimizer changes the AST or some intermediary program representation produced by the parser in order to make the output program faster or smaller. The code generator is responsible for generating code in the target language.

Traditional compiler construction courses present most or even all aspects of every compilation phase before moving on to the next one. As a consequence, students feel lost in details, losing the big compiler picture. Only at the middle or at the end of the course that a complete compiler emerges ? sometimes a complete compiler is never presented. Students usually ask the question "why am I learning this?". Since they do not know the whole, they do not understand why the parts are necessary.

This article presents the details of a different compiler construction course which has been taught every year since 2002 at the Computer Science Department of the Federal University of São Carlos, Brazil. The subject is introduced through examples of increasing complexity, starting with a very simple expression grammar and finishing with a complete compiler of a language similar to Pascal. The next section explains how these examples are presented to the students. The last section concludes.

## 2. The Course Outline

The compiler construction course is taught in one semester with sixteen weeks, three of which are reserved for examinations. There are four consecutive 50-minutes classes a week. Most of the students are in the fifth (Computer Science) or seventh (Computing Engineering) semester of their courses. Now that we described the context, the course outline can be presented.

The course is divided in two parts, each one eight-weeks long. The first one is very practical. We teach how to build compilers without worrying in proving why the techniques presented work. In the second part, we teach the theory behind compiler construction. This inversion is made on purpose. The objective is to introduce the subject as fast as possible to enable students to build a simple compiler in the first month. Students do not miss the theory since it is intuitively clear that the parsing method used, recursive descendent parsing, works.

The first course part uses ten compilers made using recursive descendent analysis and five compilers made using CUP/Jlex [1] [8]. The first ten compilers were made in Java without the help of any tool. All of the

compilers [6] and a manual [7] explaining them are available to the students . In the first day of the course, in four 50-minutes classes, the following topics are seen:

- the definition of a compiler;
- the compiler cousins: where compiler techniques may be employed;
- the phases of a compiler (lexical analysis, syntax analysis, semantic analysis, and code generation and optimization).

In the second course day, the first five compilers are introduced. In these compilers, lexical analysis is very simple, trivial. We concentrate in the more interesting parsing and code generation phases. A lexical analyzer can be made without any sophisticated technique and we chose to show more complex lexical analyzers later on. Code generation is rapidly presented to catch the imagination of the students – it seems something magical to automatically transform one language into another. None of the first five compilers needs a symbol table – there is no semantic analysis. But the abstract syntax tree is built for the 5th compiler. Compilers 6 to 10 are seen one a day, approximately. All compilers were made in Java.

The ten compilers made using the recursive descendent parsing method are presented in the following paragraphs. The important topics introduced with each of them are discussed.

Compilers 1 through 5 use the same language whose grammar is

```
Expr ::= '(' Oper Expr Expr ')'
       | Number
Oper ::= '+' | '-'
Number ::= '0' | '1' | ... | '9'
```

Legal "programs" in this language are expressions like

```
1
(+ 1 2)
(- (+ 5 2) 4)
```

Using this grammar we teach how to make a lexical and a syntax analyzer in Java. The lexical analyzer is very simple: it skips white spaces and returns the next character – note that all terminals are one-character tokens.

The parser is not difficult either. It is composed by methods of class `Compiler`, which also contains one method (`nextToken`) for lexical analysis. Class `Compiler` contains one method for each grammar rule – `expr`, `oper`, and `number`. Each parser method is responsible for analyzing the corresponding grammar rule and returns nothing (`void`). Some examples with even simpler grammars are shown and the subject is not difficult to be understood by the students. Compiler 1 does not generate code – it is in fact just a parser.

There is a method `error()` which is called whenever a lexical or parser error is found. This method prints a message and terminates the program. Note the whole compiler is in class `Compiler`.

Compiler 2 generates code using the simplest possible way: by adding print (`System.out.println`) statements to the parser code. The target language is C, which means code generation is very simple. For example, "`(+ 1 2)`" produces "`(1 + 2)`".

Compiler 3 generates assembly code. A stack-based virtual machine is used. This compiler is not too different from the previous one. It shows that non-optimized code generation to assembly is generally easy to do.

Compiler 4 evaluates the value of the expression at compile time. It shows the very basic techniques of interpreters – to evaluate an expression is to interpret it. Each parser method (`expr` and `number`) but `oper` returns the value of the expression analyzed by the corresponding rule.

Compiler 5 builds the AST (abstract syntax tree) for the input expression. The AST is a set of objects representing the input. These objects are instances of AST classes `CompositeExpr` and `NumberExpr`. Class `CompositeExpr` represents an expression with an operator, like "`(+ 1 2)`". Class `NumberExpr` represents a number. These classes inherit from abstract class `Expr` which has an abstract `genC` method. Methods `expr` and `number` of class `Compiler` return objects of the AST corresponding to the expression they analyze:

```
Expr expr() { ... }
NumberExpr number() { ... }
```

The course material [7] explains the two reasons `CompositeExpr` and `NumberExpr` must inherit from `Expr`:

- method `expr` of the parser returns either a `CompositeExpr` or a `NumberExpr` object. If the rule `Expr ::= '(' Oper Expr Expr ')'` is chosen, instead of `Expr ::= Number`, method `expr` returns a `CompositeExpr` object. Otherwise it returns an object of `NumberExpr`. Therefore the return type of `expr` must be a common superclass of `CompositeExpr` and `NumberExpr`. We created `Expr` for that;
- class `CompositeExpr` has three instance variables:

  ```
  char oper;
  Expr left, right;
  ```

  `left` and `right` are pointers to the left and right expressions of a composite expression. Both left

and right must be able to point either to a `CompositeExpr` or to a `NumberExpr` object. Then their types should be `Expr`, a common superclass.

Code generation is removed from methods `expr` and `number` of class `Compiler` and placed in `genC` methods of the AST. There is an abstract method `genC` in `Expr` and concrete `genC` methods in `CompositeExpr` and `NumberExpr`. The top-level parser method is method `compile` of class `Compiler`. It returns an object of type `Expr` whose real class at runtime is one of the `Expr` subclasses. By sending the `genC` message to this object, a method of `CompositeExpr` or `NumberExpr` is called. This illustrates polymorphism in Java.

During the course, we try to teach as much object-oriented programming as possible. And there is plenty of opportunities for that in the design of the AST classes. All the important aspects of object-oriented programming are explored: classes, inheritance, and polymorphism.

Compiler 6 uses a language that supports the declaration of variables. A typical program would be

```
a = 1 b = 7 : (+ a b)
```

This language demands some semantic analysis, since a variable may be declared twice and a non-declared variable may be used in the expression following the colon. However, we do not introduce semantic analysis in this compiler, which is only useful for showing new AST classes (`Program`, `Variable`, `VariableExpr`) and a more complete code generation to C ? now the output can be compiled by a C compiler and executed. The previous compilers generate just the expression in C, without the `main` function.

Again, there are new opportunities to teach object-oriented programming here. The declaration of a variable,[1] "`b = 7`", is represented by an object of class `Variable`. Method `genC` of this class generates "`int b = 7;`" for the statement "`b = 7`". When a variable appears in an expression, we should not use class `Variable` of the AST to represent it. Code generation for variable `b` when it appears in the expression "`(+ a b)`" is different from the code generation for the declaration of `b`. Therefore variable `b` in an expression should be represented by another class, which is `VariableExpr`. The grammar of language 6[2] defines the following rule:

```
Expr ::= '(' Oper Expr Expr ')'
       | Number
       | Variable
```

[1] Variables are in fact constants, since their values cannot be changed.

[2] Source language of compiler 6.

Method `expr` of the parser returns a `VariableExpr` object when the last option is chosen (corresponding to `Expr ::= Variable`). That means `VariableExpr` should inherit from `Expr`.

Compiler 7 uses the same grammar as language 6. It evaluates the expression through methods `eval` added to several AST classes. Each AST class represents part of an expression and the `eval` method of that class returns the value of that part.

A hash table plays the role of a symbol table and is used to keep the values of the variables. At the declaration of a variable, the pair (name, value) is inserted at the table. When a variable is found in the expression, its value is retrieved from the hash table. The compiler checks if a variable is being declared twice and if it is declared before used.

This compiler is another nice introduction to interpretation (the other is compiler 4). Instead of generating code to a virtual machine and interpreting it, this compiler interprets the AST directly, an easy way of building an interpreter.

Compiler 8 introduces new grammar rules with long terminals like `if`, `then`, and `begin` ? all previous terminals had just one character. There are numbers with more than one digit and comments from `//` till the end of the line. The language supports declaration of variables, `if`, `read`, and `write` statements. There are great changes in the lexical analyzer which now uses integers to represent terminals (previous lexical analyzers used the one-character terminals themselves). There are new AST classes: `AssignmentStatement`, `IfStatement`, `ReadStatement`, and `WriteStatement`. All of them are subclasses of the abstract class `Statement`, which declares an abstract `genC` method.

Compiler 9 introduces several novelties related to object-oriented programming. A class `Lexer` is created for lexical analysis ? method `nextToken` is moved to it. A class `CompilerError` is created just for error signaling. Class Compiler has the parsing methods. There are just one object of each of classes `Compiler`, `Lexer`, and `CompilerError`. Each one references the other two. Variables have types. There are types `integer`, `boolean`, and `char`. Each type is represented by a class of the AST and all type classes inherit from abstract class `Type`. At runtime, only one object of each of the type classes is created ? there is no need to create more than one. All objects representing, for example, type `char` would be equal to each other. Types introduce a lot of semantic checking: the if expression must have type `boolean`, the left and right-hand side of an assignment must have the same type, and so on.

Compiler 10 uses a grammar with several new rules. The language resembles Pascal and supports procedures, functions, and loop statements. The symbol table needs to be improved since there are global subroutines and local variables and parameters ? it is necessary to use two hash tables, one for each scope. We could have used a more efficient hash table but we did not because this would be a distraction from the main goals of the course.

In this compiler there are new opportunities for semantic analysis and code generation. For example, when a procedure is called the compiler should check the number and types of the arguments. Code generation is not difficult because procedure and function declarations are translated to function declarations in C ? the difficulties of the subject are masked by the target language.

The abstract syntax trees used in compilers 8-10 are not too abstract. We chose to add to them more information than they usually have. All identifiers are represented in the AST classes by pointers to objects. For example, a statement "`b = 1`" is represented by an object of the AST class `AssignmentStatement`. This class has an instance variable of type `Variable`. An object of this class references the object of class `Variable` that represents "`b`". Usually "b" would be represented by string "b". This way of building the AST adds more object-oriented programming to the compiler construction.

After studying the ten compilers made using the method of recursive descendent parsing, we present five compilers made using CUP/JLex[1, 5]. These tools are the equivalent to YACC/Lex for Java. CUP is a parser generator and JLex creates a lexical analyzer from a description of the terminals. These five compilers are the equivalent of the first five compilers made by hand using recursive descendent parsing.

The second part of the course deals with theory of compiler construction. The students learn why the recursive descendent parsing method works. It is interesting to note that, while studying the first ten compilers, the students have the intuition that the method works. They ask questions like "what if we had a rule A ::= B | C and both B and C start with the same terminal?". These questions show a correct intuitive understanding of the method.

## 3. Conclusion

In the course described in this paper, the concepts and techniques of compiler construction are introduced incrementally through a series of ten compilers. The incremental and smooth additions of features to each language make it *relatively* easy to learn the subject. The most attractive parts, parsing and code generation, are introduced in the very first compilers, motivating the students. In the course material, at the end of each compiler description there are exercises relative to the new techniques presented in that compiler. In the classes, exercises are given after every new topic to involve students with the subject.

In our course, students learn how to build a complete compiler in the second course day. They get the big picture of the subject immediately. All of the material that follows brings only refinements (although important) to the compilers taught in this day. Ghuloum [4] proposes a similar teaching method using Scheme, although his compilers are much more sophisticated than we thought could be taught in an undergraduate course. It is interesting to note that the critics he makes on traditional courses are virtually the same as ours.

By presenting the theory after and not before, we create suspense on the reasons that make the recursive descendent parsing method works. The compilers are a motivation to study the theory. It is worth remembering that it is in compiler construction that theory and practice meet each other. Without theory, there would be no systematic technique for compiler construction.

Several articles discuss compiler construction courses [2], [3], [8], [10]. However, these articles focus on the student assignments, the compilers the students should implement. It can be a compiler for a) a small ad-hoc language, b) a subset of a known language, c) an object-oriented, functional, or logic language, d) a real language in which the Professor supplies part of the code (a "fill in the blanks" approach). Or the assignments can be a mixture of the above. In this article we stress another topic, the teaching of compiler construction itself. By presenting practice before theory we get the students interested in the subject. By presenting the compiler techniques in small steps we keep them interested because the increments from one compiler to the next are not that difficult to follow.

This course is followed in the subsequent semester by a compiler laboratory course. In it, students build a complete compiler for a small object-oriented language. This language is a subset of Java called Krakatoa (a very significant name indeed). In fact, Krakatoa can be considered the smallest Java subset that is object-oriented. It has everything necessary to be considered object-oriented and nothing more.

Code generation is made to C with all the complexities brought by inheritance, polymorphism, and message sends to variables, this, and super. Although we present a paper that describes how to generate code, this is not a trivial task to the students.

Since this laboratory compiler course has no expositive classes, the Krakatoa compiler is made only with the practical experience and theoretical basis students acquired in the first course, the one described in this paper. This is, in our opinion, the greater evidence of the success of this "incremental" course method.

All the course material is available on the Internet [6].

## Bibliography


[1] Ananian, C. JLex: A lexical analyzer generator for Java. http://www.cs.princeton.edu/~appel/modern/java/JLex, 2003.

[2] Baldwin, D. A compiler for teaching about compilers. In *Proceedings of the 34th SIGCSE technical symposium on Computer science education*. (Reno, Nevada, Feb. 19-23, 2003), 220-223.

[3] Coon, L. A sequence of lab exercises for an introductory compiler construction course. *SIGCSE Bulletin* 28, 3 (Sep. 1996), 60-64.

[4] Ghuloum, Abdulaziz. An incremental approach to compiler construction. In Proceedings of the 2006 Scheme and Functional Programming Workshop, Portland, OR, 2006.

[5] Grune, D., Bal, H., Jacobs, J.H. and Langendoen, K. *Modern Compiler Design*. John Wiley & Sons, 2000.

[6] Guimarães, José de O. Compilers used in the course. Available at http://www.dc.ufscar.br/~jose/courses/cc-en.htm.

[7] Guimarães, José de O. Learning Compiler Construction by Examples. Course material. Available at http://www.dc.ufscar.br/~jose/courses/cc-en.htm.

[8] Hudson, S. CUP parser generator for Java. http://www.cs.princeton.edu/~appel/modern/java/CUP.

[9] Neff, N. OO Design in compiling an OO language. In *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*. (New Orleans, Louisiana, Mar. 24-28, 1999), 326-330.

[10] Tempte, M. A compiler construction project for an object-oriented language. In *Proceedings of the twenty-third SIGCSE technical symposium on Computer science education*. (Kansas City, Missouri, Mar. 5-6, 1992), 138-141.