IJCRT.ORG ISSN : 2320-2882

INTERNATIONAL JOURNAL OF CREATIVE RESEARCH THOUGHTS (IJCRT)

An International Open Access, Peer-reviewed, Refereed Journal

COMPILERS IN UNIVERSITY SYLLABUS

Abhishek Kundu[1],Gaurav Divtelwar[2]

[1]Assistant Professor, Kavikulguru Institute of Technology and Science, Ramtek, Maharastra, India

[2]Assistant Professor, Kavikulguru Institute of Technology and Science, Ramtek, Maharastra, India

Abstract: In traditional university compiler syllabus, all phases of compiler is studied in detail in each unit before moving on to the next one. This makes the students lose the big picture of the subject. The course format should be updated in such a way which incrementally introduced through ten compilers of increasingly complexity. The first phase of compiler is syntax analyzer of any simple language. The last one is a complete compiler of a Pascal-like language. Students from this course learn how to build compilers faster than the usual.

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

1. Introduction

There are number of reasons why students usually consider difficult courses in 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 own particularities, algorithms, techniques, and tricks of the trade.

A compiler transforms a programme written in source language S into another programme written in target language T as output. The lexical analyzer arranges the input characters from the language S into units known as tokens. Each language terminal is a token that has an integer constant associated with it. The tokens are passed into the syntax analyzer (parser), which then determines if the source programme adheres to the S grammar. An abstract syntax tree (AST) of the source programme may be created by the parser. A data structure known as an AST is used to represent all the major input components. The source software contains all of the pertinent data.

This type of validation is handled by the semantic analysis. Generally speaking, the semantic analyzer is made up of numerous pieces of code dispersed across the parser. The code optimizer modifies the AST or another intermediate programme representation created by the parser to speed up or reduce the size of the output programme. The task of writing code in the target language falls to the code generator.

Before going on to the next compilation phase, traditional compiler construction courses present the majority, if not all, of each compilation phase. As a result, pupils lose sight of the larger picture and become lost in the specifics. A complete compiler only appears in the middle or at the end of the course? There are situations when a whole compiler is never shown.

2. The Curriculum Outline

The twelve-week compiler construction course is taught across seventh semester, with two of those weeks set aside for exams. Every week, there are four consecutive 55-minute lessons. The majority of the students are enrolled in courses in their fifth (computer science) or seventh (computer engineering) semesters. The course overview can now be offered after the context has been explained.

The course is split into two sections, each lasting eight weeks. The first one is quite useful. We don't concern ourselves with explaining why the methods we describe work; instead, we teach how to develop compilers. We cover the theory underlying compiler construction in the second section. This inversion is a deliberate choice. The goal is to teach the topic as quickly as possible.

Five compilers produced with CUP/Jlex and ten made with recursive descendent analysis are used in the first course portion, respectively. Without using any tools, the first ten Java compilers were created. The students have access to all of the compilers [6] and a manual [7] describing them. The following topics are covered on the first day of the course in four 55-minute classes: the definition of a compiler; the compiler cousins: situations in which compiler techniques may be used; and the stages of a compiler (lexical analysis, syntax analysis, semantic analysis, and code generation and optimization).

The first five compilers are introduced on the second course day. Lexical analysis is quite basic and trivial with these compilers. We focus on the code generation and more intriguing parsing stages. A lexical analyzer may be created using simple methods, but we will demonstrate more advanced lexical analyzers later. To capture the students' attention, code generation is quickly presented? The automatic conversion of one language to another looks amazing. The first five compilers don't require a symbol table, right? Semantic analysis is absent. However, the fifth compiler is designed for the abstract syntax tree. One of compilers 6 to 10 is spotted every day, on average. Every compiler was created in Java.

The language used by compilers 1 through 5 has the same grammar as

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

In this language, statements like are considered legal "programmes."

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

We demonstrate how to create a Java lexical and syntax analyzer using this grammar. The lexical analyzer is quite basic; it returns the following character after skipping over white spaces. Tokens with one character are used for all terminals.

The parser is also not tough. It is made up of functions from the class Compiler, which also has a method (nextToken) for lexical analysis. There is a method in Class Compiler for each grammatical rule.

In order to analyse the relevant grammatical rule, each parser method must return nothing (void). The subject is not difficult for the pupils to understand, and several examples with even simpler grammars are provided. No code is produced by compiler 1? It is only a parser, after all.

There is a method error() when a lexical or parser mistake is discovered, this function is invoked. This technique ends the programme and prints a message. Take note that class Compiler contains the whole compiler.

The simplest method for generating code is used by Compiler 2: it just adds print (System.out.println) instructions to the parser code. Since C is the intended language, code creation is fairly straightforward. For example, “(+ 1 2)” produces “(1 + 2)”.

Assembler code is produced by compiler 3. The virtual machine is stack-based. The old compiler and this one are not drastically different. It demonstrates how simple it is to generate non-optimized code for assembly.
At the time of compilation, compiler 4 assesses the expression's value. It displays the very fundamental methods used by interpreters. An expression is interpreted when it is evaluated.
Except for oper, each parser method (expr and number) returns the result of the related rule's analysis of the expression.
The AST (abstract syntax tree) for the input expression is created by compiler 5. The input is represented by a collection of items in the AST. These objects are examples of the AST classes NumberExpr and CompositeExpr. Expressions with operators, such as "(+ 1 2)," are represented by the class CompositeExpr. A number is represented by Class NumberExpr. These classes derive from the abstract class Expr, which has a method called abstractGenC. methods, class numbers, and expr Compilers return items from the AST that correspond to the analysed expression:
Expr. expr () NumberExpr number(), NumberExpr...,...
The two explanations for why CompositeExpr and NumberExpr must inherit from Expr are provided in the course materials [7]:
The parser's method expr produces an object that is either a CompositeExpr or a NumberExpr. Method expr produces a CompositeExpr object if the rule Expr::= '(' Oper Expr Expr ')' is used in place of Expr::= Number.
If not, it returns a NumberExpr object.
As a result, CompositeExpr and NumberExpr must have a similar superclass as the return type for expr. For that, we developed Expr, which contains three instance variables of class CompositeExpr:
char oper; Expr left, right; left and right are pointers to a composite expression's left and right expressions, respectively. Neither left nor right may point to a NumberExpr object; only a CompositeExpr object is permitted.
Then Expr, a widespread superclass, should be their types.
Variable declaration is supported by the language that Compiler 6 uses. Typically, a = 1 b = 7: (+ a b) would be the programme.
Since a variable may be declared twice and a non-declared variable may be utilised in the statement that comes after the colon, this language necessitates some semantic examination.
However, as semantic analysis is mainly beneficial for demonstrating new AST classes (Program, Variable, and VariableExpr) and a more thorough code generation to C, we do not implement it in this compiler. Now a C compiler can compile the output and run it. The main function was not produced by the earlier compilers; just the C expression was.
Language 6 and Compiler 7 both employ the same grammar.
Through eval methods that have been introduced to numerous AST classes, it evaluates the expression. Every AST class represents a different piece of an expression, and each class's eval function returns the value of that piece.
The values of the variables are stored in a hash table, which serves as a symbol table. The name and value pair are added to the database during the variable declaration process. A variable's value is obtained from the hash table when it is discovered in the expression. A variable's double declaration and whether it was declared before usage are both flagged by the compiler.
Another good introduction to interpretation is this compiler (the other is compiler 4).
In Compiler 8, new grammatical rules with lengthy terminals like "if," "then," and "begin" are introduced. There was just one character on each terminal before. There are comments beginning with / and numerals with more than one digit. Variable declaration, if, read, and write statements are supported in the language.

The lexical analyzer has undergone significant modifications, switching from using one-character terminals itself to representing terminals as integers. AssignmentStatement, IfStatement, ReadStatement, and WriteStatement are new AST classes. Each of these is a subclass of the abstract Statement class, which is where the abstract genC function is declared.

There are various new features in Compiler 9 that are specifically linked to OO programming. For lexical analysis, a class called Lexer is developed. It now has the function nextToken. CompilerError is a class built specifically for error reporting.

The parsing techniques are in the Class Compiler. The classes Compiler, Lexer, and CompilerError each have a single object. The other two are mentioned in each one.

Types exist for variables. There are three types: char, boolean, and integer. All type classes derive from the abstract class Type, and each type is represented by a different class in the AST. Is it true that just one instance of each type class's object is produced at runtime? No need to produce more ones. For instance, all objects representing the type char would be identical to one another. Types include a lot of information.

The grammar used by Compiler 10 has a number of new rules. The language, which has features like functions, loop statements, and procedures, is similar to Pascal. Since there are local variables and arguments as well as global subroutines, the symbol table needs to be updated. Two hash tables must be used, one for each scope. We could have utilised a more effective hash table, but we decided against it since it would have taken our attention away from the course's key objectives.

3. Conclusion

Through a series of 10 compilers, the principles and methods of compiler development are gradually presented in the course detailed in this article. Learning the topic is rather simple due to the gradual and steady addition of features to each language. The very first compilers introduce the most appealing components, parsing and code creation, inspiring the pupils. There are exercises related to the new approaches introduced in each compiler's explanation towards the end of the course materials. Exercises are offered in class following each new topic to get students more involved in the material.

Students in our programme learn how to construct entire compiler on day two of the training. They obtain large picture of the topic right away. All the information that comes next just offers improvements (although important) to the compilers being taught today. Ghuloum (4) suggests a same teaching strategy utilising Scheme, although his Compilers are far more advanced than we initially believed. might be covered in a college course. It's fascinating.

to emphasise that his criticisms of conventional courses are almost identical to ours.

Courses on compiler construction are covered in a number of articles [2, [3, [8,] [10]. These articles, however, are primarily concerned with the projects that students must complete and the compilers they must use. It may be a compiler for one of the following: a) a limited, ad hoc language; b) a subset of a widely used language; c) an object-oriented, functional, or logic language; d) a genuine language in which the professor gives some of the code (a "fill in the gaps" method). Or the tasks might be a combination of the aforementioned. We also emphasise the teaching of compiler building in this essay. We pique the attention of the students in the material by giving experience before theory. The increments from one compiler to the next keep them interested, thus we offer the compiler concepts in incremental chunks. The semester after this course, a compiler laboratory course is offered. Students create a full compiler for a compact object-oriented language in it.

A subset of Java by the name of Krakatoa was used to create this language. In fact, Krakatoa may be thought of as the smallest object-oriented subset of Java.

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, Ne vada, 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/ccen.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 twentythird SIGCSE technical symposium on Computer science education. (Kansas City, Missouri, Mar. 5-6, 1992), 138- 141