# A PROGRAMMING LANGUAGES COURSE EMPHASIZING INTERPRETERS AND OJBECT-ORIENTED DESIGN

Steve Carr
Department of Computer Science
Western Michigan University
1903 W. Michigan Ave.
Kalamazoo MI 49008-5466
steve.carr@wmich.edu

**ABSTRACT**

Although a course in programming languages is fundamental to the discipline of computer science, much debate still exists over the proper approach to teaching such a course. In this paper, we present the design of a programming languages course taught at Western Michigan University. The course teaches programming language concepts using interpreters, while at the same time stressing the use of object-oriented design patterns. The result is a course that covers practical topics for all students in the context of teaching language semantics through interpreter development.

**INTRODUCTION**

Many students develop solid programming skills by the time that they reach a junior- or senior-level programming languages course. However, few completely understand the features contained within many modern programming paradigms. Most students can use a programming language, but few understand them. One of the major goals of a programming languages course is to help students understand the meaning of the fundamental programming language concepts that form the basis of the development of programming languages [4]. By developing a better understanding of what it means to state something in a programming language, a student will better understand the complexity and behavior of programming languages. This understanding, in turn, creates better programmers.

Currently, several different methods exist for teaching students the material in a programming languages course. These include:

1. A survey of various programming paradigms that illustrates the principles of each paradigm and provides example languages in each paradigm [10][11],
2. A semantics-based approach that can utilize formal semantic models or can emphasize the expression of semantics through the development of interpreters [4][12], and
3. A pragmatic approach that puts greater emphasis on the implementation of programming languages [9].

Each of these approaches has its merits. In this paper, we describe the approach taken at our university that combines method 2 with an extensive implementation emphasizing object-oriented design patterns [3][5].

**RELATED WORK**

Debray presents a compiler design course intended to make the topic relevant to students who will not design compilers [2]. He argues that the principles and techniques used in compiler design are applicable to many translation problems that students are likely to encounter. In the same way, our

course is relevant to students who will not write interpreters. They learn a deeper understanding of how programs work and also get practical experience in object-oriented design.

Hallstrom, et al., give a course that integrates languages and software engineering [6]. They introduce design patterns in a sophomore-level course where a new language (Java) is introduced to illustrate the differences between procedural and object-oriented paradigms. Our course is similar, but we introduce design patterns in an upper-level course using more complex data structures.

Friedman, et al., present the semantics of programming languages through the development of interpreters [4]. They develop interpreters in Scheme. Our approach covers many of the same topics, but uses object-oriented programming to develop the interpreters. We have chosen the object-oriented approach to stress skills that students are quite likely to encounter in the work place.

**COURSE GOALS**

At Western Michigan University, introductory object-oriented design and implementation are covered in Computer Science II. This material is used and extended in Data Structures, however, the projects in both of these courses tend to be relatively simple in their structure and depth in object-oriented design. Faculty have found that upon reaching senior design, students have not had sufficient experience in object-oriented programming concepts such as design patterns to develop reusable software effectively. Unfortunately, we neither have sufficient faculty nor room in the curriculum to teach a course devoted to object-oriented design. As a response to this need, we have designed a course in programming languages that addresses both the fundamental concepts in language design and the need for advanced instruction and implementation in object-oriented programming.

The approach we have taken in the design of our programming languages course is to develop an object-oriented interpreter for a functional programming language. This approach allows us to cover the fundamental language design concepts and at the same time stress object-oriented design through the project. The major goals of our course are:

1. Given a program language, the student will be able to state clearly the syntax and semantics of the language, and the difference between the two.
2. Given a complex data structure, the student will be able to use design patterns to implement that structure and to build and traverse that structure.
3. Given a language paradigm, the student will be able to assess the attributes of that paradigm for its strengths and weaknesses.

In the rest of this paper, we describe the course modules and project that we have designed to meet these goals.

**COURSE MODULES**

We have divided up this course into four modules: language recognition, language representation, rudimentary language semantics and language environments. This sections gives an overview of the contents of these modules.

**Module: Language Recognition**

This module focuses on the application of computation theory to programming languages. The major concepts covered in this module are: regular expressions and scanners, context-free grammars, LL(1) grammars and parsers, and parser and scanner generators.

*Module Goal*. The goal of this module is to give a practical application of computation theory in order to show the value of understanding basic computer science theory. Once this module is finished, students should be able to state the regular expressions and context free grammar for a simple programming language and understand the roles of the scanner and parser generated from those components.

*Regular Expressions and Scanners*. First, we cover regular expressions and their implementation as scanners [1]. We describe the process of going from the specification of a regular expression to the construction of a non-deterministic finite automaton (NFA), to the conversion of the NFA to a deterministic finite automaton (DFA). We currently do not cover DFA minimization. Finally, we show how to code a simple scanner from the transition function of a DFA and discuss issues that real scanners need to handle in practice.

*Grammars and Parsers.* The next part of this module discusses the specification of context-free grammars and the construction of top-down recursive descent parsers. We discuss the properties of LL(1) grammars and the transformation of grammars to LL(1) form. However, we do not require students to understand the algorithms necessary for transforming grammars to LL(1) form. Given a simple LL(1) grammar, we show how to construct a basic top-down recursive parser containing a function for each non-terminal with references to terminals on the right-hand side of a rule becoming calls to the scanner and references to non-terminals becoming function calls [1].

*Parser and Scanner Generators.* In the project described later, we do not require the hand construction of either the scanner or parser, but rather use a parser generator. Thus, we cover a parser generator such as ANTLR [8] or JavaCC [7].

*Ongoing Example.* For each of the above concepts, we use a simple calculator language (Calc) to give a practical example of how the concepts are used in specifying the syntax of a programming language. Calc has the following structure:[1]

$$
\begin{array}{lll}
C & \rightarrow & C * C \\
 & | & C \div C \\
 & | & C + C \\
 & | & C - C \\
 & | & \textbf{number}
\end{array}
$$

In class, we specify a parser and scanner using a parser generator. This specification provides a starting point for the project discussed later.

[1] ANTLR 4 is able to handle this grammar even though in its stated form it is ambiguous and not LL(*k*).

**Module: Language Representation**

The next step in developing the interpreter of a programming language is to represent a programming language as a data structure. The major practical value of this course module is to give students a reasonably complex example of object-oriented design. In addition, this module helps students map a complex problem to an abstraction, increasing their ability to deal with abstract programming concepts. The major concepts covered in this module are: inheritance, design patterns, recursively defined data and abstract syntax trees (ASTs).

*Module Goal.* Beyond learning the concepts of parse trees and ASTs, the major goal of this section is to reinforce object-oriented design by doing it. ASTs are significantly more complicated in abstraction than data structures encountered by students in previous courses. The AST for the project consists of around 50 classes with an inheritance hierarchy of up to depth 3. This presents students with a level of abstraction that is challenging and also shows the value of design patterns in constructing a reusable solution.

*Inheritance.* To begin, we give a brief review of inheritance. In our course we present inheritance in Java, although this material need not be tied to any particular language.

*Design Patterns*. Once we review inheritance, we present the concept of design patterns in software development. Students at our university have not seen design patterns previously, so it is critical for us to expose them to this concept before reaching senior design. We cover the following patterns: factory, builder, visitor and composite. These particular patterns are covered because we emphasize their use in the design of abstract syntax trees.

*Recursively Defined Data.* After design patterns, we cover recursively defined data [4]. Since the building and walking of an abstract syntax tree requires recursive programming, we use this section to emphasize how the recursive specification of data leads to a natural recursive solution to walking the data structure. We introduce a grammar for specifying tree-based data structures that is based upon tree grammars from JavaCC [7]. We use the following form for productions in specifying recursive data:

$$
\begin{array}{lll}
N & \rightarrow & \text{'\#' '(' } Y(C|N)^{+} \text{ ')'} \\
Y & \rightarrow & \text{'\$'}[a-zA-Z]^{+} \\
C & \rightarrow & '@'[a-zA-Z]^{+}
\end{array}
$$

where *N* defines a node in the data structure, *Y* defines a node type and *C* defines an atomic element (e.g., a Java built-in type). As an example consider the following definition of a binary tree.

$$
\begin{array}{lll}
B & \rightarrow & \#(\textbf{\$treenode}\ @\text{int}\ B\ B) \\
 & | & \lambda
\end{array}
$$

These rules define a tree *B* as a node of type **$treenode** containing an integer and two *B*'s, or as a null node. From this definition, we can derive the structure of a recursive routine to walk a *B* noting that the references to *B* on the right hand side of the rule indicate that a recursive call is needed.

*Abstract Syntax Trees.* The last topic of this module is an abstract syntax tree (AST). We cover the difference between parse trees and ASTs. In class, we convert a parse tree returned by the parser generator into an AST for the Calc language. The AST is defined using a tree grammar as defined previously. We illustrate the composite pattern to design an *n*-ary AST, the factory and builder patterns to build the AST, and the visitor pattern to walk the parse tree and AST. We demonstrate how a recursive data specification translates into an object-oriented recursive visitor pattern.

**Module: Rudimentary Language Semantics**

Once we have a program represented as an AST, we turn to a module on creating an interpreter for a simple language without user-defined functions. The major concepts covered in this module are: interpretation of expressions and variables, language defined operations and control flow.

*Module Goal.* The main goal of this module is to help students understand the difference between syntax and semantics. The student should understand that syntax means nothing without a context in which to evaluate it.

*Interpretation of Expressions and Variables.* In this section, we write an interpreter for the basic Calc language using a visitor pattern on recursively defined data. Then, we introduce the concept of static scope and environments by adding a form similar to **let** in Scheme. We show how the meaning of a program is dependent on the context in which it appears, and assert that the students have been unconsciously evaluating computer programs in a defined environment already. We illustrate this by showing how we can redefine language defined function names in a simple C program noting that the context of the C code must be known in order to ascertain its meaning correctly.

We next introduce the concept of an environment that maps variable names to values. Using this mapping we can develop an interpreter for the Calc language that adds a form similar **let**.

*Language Defined Operations*. After we have introduced environments as a means to get the meaning of a variable, we discuss language-defined names. We introduce the concept of a base environment and how we can abstractly represent base functions as objects. Since this concept adds complexity because of the abstraction, we also show how to add the language defined names to the grammar and add AST node types to handle them. Although the second method is inferior because it is cumbersome and restrictive to extend the language, it provides a means for students already struggling with the complexity of the object-oriented design patterns to be able to implement the meaning of language defined functions using a method already covered while building ASTs.

*Interpretation of Control Flow.* In this section, we discuss the effect of precedence on the evaluation order of expressions and show how it has a significant effect on the meaning of an expression. We also cover the evaluation of logic expressions and the effect of short-circuit evaluation. Finally, we discuss if-then-else, switch statements and loops. Beginning in this section, we no longer develop interpreters in class, but rather expect students to be able to take the discussion and implement the concepts in the project.

**Module: Language Environments**

Once we have covered the basics of interpretation, we move to more complex environment structures. The major concepts in this module are variable scoping, first-class functions, parameter passing and objects.

*Module Goal*. The main goal of this module is to help students understand how scoping mechanisms are given meaning in an interpreter. Ultimately, we believe that if a student understands how an interpreter handles program scope, we will produce a better software developer because that developer will understand what his or her program is doing with greater clarity. Understanding programming language environments helps remove the "magic" of how programs are executed.

*Static Scoping*. In this section, we introduce nested environments that implement static scoping. To handle nested function definitions, we introduce the concept of a closure and how a closure handles static scoping. We feel this concept is especially important because our students have not had exposure to first-class functions and most modern languages, including Java SE 8, implement this concept. Once we cover closures, we discuss the meaning of dynamically scoped variables.

*Parameter Passing*. In this section, we cover call-by-value, call-by-reference, call-by-value-result and call-by-name semantics. We show via example how each parameter passing semantics leads to different meanings of the same expressions. We currently do not develop different interpreters for different semantics.

*Objects*. In this section, we show how to interpret objects with inheritance and class variables using a properly structured environment. We try to emphasize the similarity with interpreting nested functions by adapting the structure of the environment to implement the scoping rules for objects. We also discuss the effects of multiple inheritance on object interpretation.

**PROJECT DESCRIPTION**

In this section, we give an overview of the project that we use to reinforce the concepts covered in this course. In our project, students use either Java or C# for implementation, although the course is not tied to any specific language.

The project is broken up into four major parts corresponding to the modules presented previously. For each part, we provide a solution that the students may use to continue with the course.

We have designed several syntaxes for a dynamically typed language that supports first-class functions and objects. Since our students use Java in many courses, this language is chosen to introduce them to functional programming and dynamic typing. Students learn about these two concepts by implementing them.

*Language Recognition.* The first project is meant to introduce students to a particular parser and scanner generator. We currently use either ANTLR [8] or JavaCC [7]. Students translate a supplied BNF

grammar into the syntax of the chosen parser generator. The goal is to be able to construct a program with a parser generator that correctly parses an input program for the chosen project language.

*Language Representation*. The second project is meant to deal with object-oriented design and design patterns. Students are given a recursive AST definition and asked to design a class structure for the AST using UML and the composite pattern. Then, students are asked to use the factory, builder and visitor patterns to translate the parse tree returned from the parser (ANTLR creates the parse tree automatically) into the designed AST. Finally, the students are asked to construct a visitor pattern to walk the AST and print out the original program.

*Simple Interpreters*. The third project involves implementing an interpreter for arithmetic expressions in the input language while performing dynamic type checking. In addition, a number of language defined functions on integers, floats, strings and lists are implemented. Finally, students interpret if-then-else constructs and a form of the Scheme **cond**-expression.

*Language Environments*. The final project involves interpretation of function definitions, function calls, first-class functions and nested **let**-expressions.

*Discussion*. This project combines both language semantics and object-oriented design. Although most students will not write interpreters, the skills learned in this project will enhance their object-oriented programming ability.

The first and third projects are not too difficult. The second project is very difficult as it is a large step in abstraction since it combines a large number of classes in the class hierarchy and the use of design patterns. We have experimented with breaking that project into many subparts with intermediate due dates. Students grossly underestimate the effort involved in this project and need to be kept on track. The final project is not as difficult from a programming point of view as the second, but intellectually, the understanding of environments poses a significant hurdle. Once students can understand closures, the coding is not too difficult.

**TEACHING METHODOLOGY**

When we teach this class, we try to incorporate active learning. We are in the process of developing several 10-15 minute video lectures that introduce many of the topics. We have done this for about 8 topics so far. Students watch the video and take a short quiz at the end. Then, in class, we solve homework problems related to the lecture and extended concepts. Even when we spend significant time on lecture material, we often code solutions to the concepts or solve example problems as a significant portion of the class.

Student evaluations have been positive toward this format of the course. In the future, we plan to incorporate more of the flipped classroom so that students are actively engaged.

**CONCLUSIONS**

We have presented a programming languages course emphasizing interpreters and object-oriented

design. Students learn the meaning of programming languages while getting significant practice in computation theory, data structures, and object-oriented design patterns. The result is a course that covers practical concepts for those who most likely will never build an interpreter but have a need to understand how programming languages work.

Once we have finished the development of the delivery of the course, we plan to assess its impact on student learning. We hope through assessment to show that the approach we have taken has improved student understanding of object-oriented programming.

**REFERENCES**


[1] K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kauffman, second edition, 2012.
[2] S. Debray. Making compiler design relevant for students who will (most likely) never design a compiler. *In Proceedings of the 33rd Annual ACM SIGCSE Technical Symposium*, pages 341--345, Covington, KY, Feb. 2002.
[3] Eric Freeman and Elisabeth Freeman. *Head First Design Patterns*. O'Reilly & Associates, Inc., 2004.
[4] D. P. Friedman, M. Wand, and C. T. Haynes. *Essentials of Programming Languages*. MIT Press, second edition, 2001.
[5] E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
[6] J. O. Hallstrom, C. Hochrine, J. Sorber, and M. Sitaraman. An ACM 2013 exemplar course integrating fundamentals, languages and software engineering. In *Proceedings of the 45th Annual ACM SIGCSE Technical Symposium*, Atlanta, GA, Mar. 2014.
[7] JavaCC. https://java.net/projects/javacc.
[8] T. Parr. *The Definitive ANTLR 4 Reference*. The Pragmatic Bookshelf, 2013.
[9] M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, 2000.
[10] R.W. Sebesta. *Concepts of Programming Languages*. Addison Wesley, 2005.
[11] R. Sethi. *Programming Languages: Concepts and Constructs*. Addison Wesley, 1996.
[12] A. Tucker and R. Noonan. *Programming Languages: Principles and Paradigms*. McGraw-Hill, 2002.