

INCORPORATING ADVANCED PROGRAMMING TECHNIQUES IN THE COMPUTER INFORMATION SYSTEMS CURRICULUM

Charles S. Saxon, Eastern Michigan University, charles.saxon@emich.edu

ABSTRACT

Incorporating advanced programming techniques in the Computer Information Systems curriculum is desirable even though the students' employment prospects in the computer industry are changing. Finding suitable vehicles for teaching advanced programming techniques is difficult since most problems that require such techniques are complicated and explaining the problems involves extensive class time. An interpreter for a small language and a graphic user interface development system are presented here as problems requiring minimal explanation time yet requiring many advanced programming techniques. These problems can be solved using object-oriented techniques and design patterns and without the use of special software or advanced language theory.

Keywords: object-oriented programming, design patterns, parsing, interpreting, algorithms, languages, program design

INTRODUCTION

The amount of technical information one needs to be competent in the area of Computer Information Systems is ever increasing. However, in many educational institutions in the United States, there is administrative pressure to reduce the number of credit hours required to obtain degrees. Therefore it is necessary to reexamine the ways in which computer techniques, especially programming techniques, are presented to students. Although students are typically exposed to computers long before they arrive at college, they are seldom taught programming techniques that have been used for over 50 years to solve problems found in nontrivial computer applications. It may be that few Computer Information Students will ever be employed as programmers, but it is hard to believe that designers and analysts can optimally specify how applications will function if they lack the knowledge of how those applications can be implemented.

The almost universal presence of libraries of data structures such as stacks and queues makes it questionable whether a traditional data structure class can still exist in the programming curriculum. Today it seems much more important that all students preparing for computer careers study object-oriented design and programming techniques rather than the implementation of standard data structures. Additionally there must room in the curriculum for studying design patterns (2) and how they can be used to program at a higher level. There is a substantial problem involved in the teaching of nontrivial programming tasks, it is difficult to find problems that require design patterns and other advanced programming techniques that can be explained in a few sentences. If the problem requires extensive explanation, that explanation takes valuable class time from the programming techniques being taught.

This paper describes how the problem of implementing a small language and a language development system with a graphic user interface can be used to introduce several design patterns and programming techniques in a second or third programming class. Because of the students' familiarity with compilers and program development systems in beginning programming classes, the amount of time spent explaining the problem to be solved is reduced to

a minimum. While the students must be made aware of some aspects of formal language theory, this can be minimized by carefully choosing the language to be implemented. Teaching a tiny amount of language theory can be justified by indicating how the general problem of language translation can be applied to numerous practical situations. Recently the need to include language processing techniques in the Computer Information Systems curriculum has increased because of the common requirement that programs parse and interpret XML statements.

The problem of designing and implementing a recursive descent parser/tree builder/interpreter for a small language (5) has been advanced previously as one meeting many of the goals described above. If the problem is expanded to include a program development environment with a graphic user interface (GUI) many additional techniques can be included. Because of the similarities to the tools the students have been using for program development, the problem specifications are easily conveyed. The techniques that can be involved in the implementation of a small language and a graphic development environment are described here. A system such as this could be built in any GUI environment using any object-oriented language such as C++, Java, or C#.

A LANGUAGE DEVELOPMENT SYSTEM

As a specific example of the kind of problem that could be used to introduce the techniques described here, a language processor and development system written in C# and developed using Microsoft's Visual Studio .NET is shown. A GUI text editor forms the basis for the program. This includes a place for the user to enter the program by opening a file or entering it from the keyboard. Buttons and pull-down menus are provided for running the parser/tree builder and for executing the program after the tree is built. Figure 1 shows the program's user interface. The pannel on the right is a text box where the program can be entered and displayed. The pannel on the left is a tree control where the parse tree is displayed after it is built. At the top there are pull-down menus and buttons which allow the user to open and save program files, build the parse tree, execute the program continuously, execute the program one statement at a time, and reverse the last step of a single statement execution.

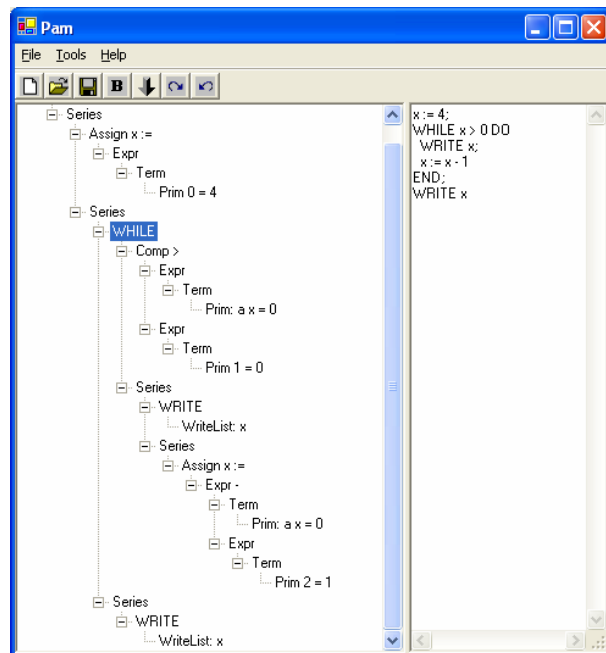


Figure 1. The User Interface

The example described here uses the object-oriented recursive descent parsing and tree building method presented in (5) to implement the *Pam* language (4), a small language that can be used to program integer calculations. The Pam language has conventional assignment, conditional, and iteration statements. While the object-oriented recursive descent parsing technique is considered an important technique, substituting a different parsing and tree building scheme for the one used in this example would have little effect on the development system described here. Similarly, the C# language and the .NET Framework Class Library controls used in this example could be

replaced by another GUI program development system without changing the design patterns used.

When the program is executed one statement at a time, the parse tree and a source code listing, which is displayed in a listbox contained in a separate window, are both highlighted to show the statement currently being executed. Figure 2 shows the window where the source code listing is displayed.

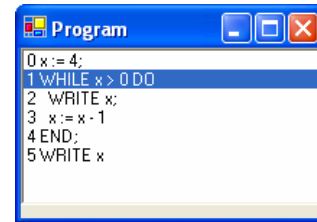


Figure 2

THE PROGRAMMING TECHNIQUES USED

Throughout this paper the term *design pattern* refers to one of the 23 design patterns described in the book (2) by the group often called the gang of four (GoF). C# implementations of these design patterns are shown in (1).

Lexical Scanning

One of the most generally useful programming techniques is the concept of taking the input in a form that already exists or a form that is the most convenient for the creator of the input and converting it into a form that is ideal for the program being written. This technique is taken to the extreme in the lexical scanner that is almost always a part of a language processor. In a language processor the lexical scanner reads the input, removes extraneous material such as white-space and comments, and constructs tokens which are passed to the parser. The tokens are typically all of the same data type but they represent a wide variety of language elements such as names, constants, key words, and punctuation. The lexical scanner relieves the parser of the problems associated with variable length elements and generally simplifies the coding of the parser. Similar techniques can be used in almost all nontrivial programs. For example, allowing the creator of the input to incorporate comments directly in the input is often highly regarded.

Recursive Descent Parsing and the Interpreter Pattern

Of the many known parsing techniques, recursive descent is the most intuitive for nearly all people. It is often used for constructing parsers by hand (3). However, many of the recursive descent parsing and tree building methods are not object-oriented, and thus fail to reinforce the object-oriented techniques presented to students in previous programming classes. An object-oriented recursive descent parsing technique which directly follows from an Extended Backus-Naur Form (EBNF) description of the syntax of the language has been shown (5). Programs developed using this technique often require the use of inheritance and polymorphism. Thus, using this method in the construction of calculators and interpreters for small computer languages not only presents the students with a new programming technique, it also reinforces the object-oriented design and implementation skills presented in previous classes. The program that incorporates the parsing operation can be a syntax checker, an interpreter, or a compiler depending upon what is done in conjunction with the parsing operation and subsequently with the resulting data structure. If nothing other than parsing is done the result is a syntax checker. If a parse tree, abstract syntax tree, or other representation of the syntax of the processed input is constructed during the parsing the result can be an interpreter or a compiler.

The Singleton Pattern

It is logically necessary that there be only one lexical scanner and one symbol table. Since many different classes may need to access the scanner and the symbol table and it would be costly to pass them as parameters, it is desirable that they are both global in scope. This can be achieved

by using a version of the singleton pattern where a public member variable of type Scanner is the one and only instantiation of the class Scanner. This fragment of C# code shows an example:

```
class Scan {  
    private Scan() { } // the only constructor  
    public int Token { get { ... } }  
    ...  
    public static Scan s = new Scan();  
}
```

There is no possibility of creating an instance of the scanner outside the class since the only constructor is private. The one and only instance is created inside the class and it is public so that the scanner can be accessed from anywhere in the program.

Logically there should only be one symbol table and it is needed throughout the program. Thus, the singleton pattern can be advantageously used for the symbol table also.

The Composite Pattern

Executing (interpreting) the program involves traveling the parse tree and taking appropriate action at each node of the tree as it is visited. This tree traversal is not the usual type where each node is visited exactly once, but instead the nodes are visited as the program is executing. For example, the nodes representing the statements in a loop are visited repeatedly until the loop ends. The parsing and tree building method used here requires that a class be defined for each production in the grammar of the language being interpreted. This results in a parse tree that consists of different kinds of nodes. Each class that represents a production in the grammar must have a method (function) that participates in the tree traveling and performs the appropriate action when that kind of node is visited. It is easy to write recursive functions to travel the tree and perform appropriate actions when the program is running continuously. However the recursive routines control the order of execution and once started the execution of the program can be paused but not reversed. To facilitate the writing of an iterator (using the iterator pattern discussed below) which will provide controlled single statement execution and undoing execution of statements already executed, the composite design pattern is used.

The Statement class is an abstract class from which each of the six different kinds of statements in the Pam language are derived. To implement the Composite design pattern, each statement class is also derived from the Composite interface. The Composite interface requires that each class implement two methods. The first method returns a reference to statements contained within the present statement. The I/O statements and the assignment statement cannot contain other statements, so they always return null. The **if** statement which can contain two groups of statements, one group to execute when the condition is true and the other group to execute when the condition is false, will return a reference to the appropriate group depending upon the state of the condition. The repeating statements, **while** and **to**, always return a reference to the one group of statements they contain. The second method of the Composite interface reports on a statement's capability for repeating and the status of the repetition if the statement has that ability. The **if**, assignment, and I/O statements have no repeating ability, so they always return a code indicating that. The repeating statements return one code if the repetition is finished and a different code if the looping continues.

The Iterator Pattern

As stated previously, it is easy to write recursive routines to travel the parse tree, but the recursive routines control the order of execution and once started the execution of the program

can be paused but not easily reversed. If the tree traveling process is done by iteration rather than recursion, program execution is more easily controlled; this also facilitates programming an undo capability. An iterator class for traveling the parse tree in the order in which the statements are executed that provides methods for beginning the traversal, progressing to the next node, recessing to the previous node, obtaining a reference to the current node, and telling when the traversal is finished facilitates the traversal process and simplifies single statement execution of the program being interpreted. For single statement execution, it is not necessary to travel the entire tree, the parts of the tree associated with expressions and conditions do not contain any statements in this language, and thus they need not be traversed.

The Command Pattern

By applying the command pattern to encapsulate the execution of the statements the recording and undoing of statements is facilitated. This makes it possible to execute a statement using the single step facility, and then use the undo facility to reverse the execution and restore the program to the state it was in before the statement was executed. In the system shown here the undo facility is only capable of going back to the previous statement, but if a record had been kept of all the statements previously executed it would be possible to revert to any previous state by repeatedly using undo. Since all input is from the keyboard, undoing a **read** statement just requests new input from the user, it does not restore variables to their previous values. Nothing is done when a **write** statement is undone. When an assignment statement is undone the variable assigned to is restored to its previous value. Undoing an **if** or **while** statement does nothing (the iterator backing up will cause the condition to be tested again and the outcome of the condition test might be different if values in the symbol table have changed). The **to** statement uses a counter to determine the number of times the statements in its body have been executed. Undoing a **to** statement reduces the counter so that the repetition will occur one more time.

The Observer Pattern

When the program is being executed one statement at a time, it is desirable to inform the user where in the parse tree and where in the statement listing the statement being executed currently is located. This can be done by highlighting the node in the parse tree that represents the statement currently executing and highlighting the line in the program listing that contains the current statement. To facilitate highlighting a line of code, the program is copied from the textbox on the form to a list box which is shown on another form. The coordination of the highlighting is accomplished by using the observer design pattern. Both of the classes associated with the forms are registered as observers and are thus notified when the current statement changes. The first form is both an observer and a subject since the menu item and toolbar button which trigger single step execution are located on the form and the tree control is also on that form.

CONCLUSION

It is important that students in Computer Information Systems programs become proficient in system implementation and programming techniques even if they will never be employed as programmers. By modifying the advanced programming classes in the curriculum, programming techniques can be taught that will influence the way students will think about problem solutions and the entire program development process. This in turn will influence the students' abilities to make significant contributions to the solutions of complex problems regardless of their roles in the process.

REFERENCES

1. Cooper, J. W., (2003) *C# Design Patterns: A Tutorial*. Addison-Wesley.
2. Gamma, E., R. Helm, R. Johnson, and J. Vlissides, (1995) *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
3. Louden, K., (2003) *Programming Languages, Principles and Practice*. 2d ed., Brooks/Cole, Pacific Grove, CA.
4. Pagan, F., (1981) *Formal Specification of Programming Languages: A Panoramic Primer*. Prentice-Hall.
5. Saxon, C. S., (2003) Object-Oriented Recursive Descent Parsing in C#. ACM SIGCSE Bulletin, *inroads*, 2003 December issue, Volume 35, Number 4, pp. 82-85.