THE INTERPRETER PATTERN

Some programs benefit from having a language to describe operations they can perform. The Interpreter pattern generally describes defining a grammar for that language and using that grammar to interpret statements in that language.

Motivation

When a program presents a number of different, but somewhat similar cases it can deal with, it can be advantageous to use a simple language to describe these cases and then have the program interpret that language. Such cases can be as simple as the sort of Macro language recording facilities a number of office suite programs provide, or as complex as Visual Basic for Applications (VBA). VBA is not only included in Microsoft Office products, but can be embedded in any number of third party products quite simply.

One of the problems we must deal with is how to recognize when a language can be helpful. The Macro language recorder simply records menu and keystroke operations for later playback and just barely qualifies as a language; it may not actually have a written form or grammar. Languages such as VBA, on the other hand, are quite complex, but are far beyond the capabilities of the individual application developer. Further, embedding commercial languages such as VBA, Java or SmallTalk usually require substantial licensing fees, which make them less attractive to all but the largest developers.

Applicability

As the *SmallTalk Companion* notes, recognizing cases where an Interpreter can be helpful is much of the problem, and programmers without formal language/compiler training frequently overlook this approach. There are not large numbers of such cases, but there are two general places where languages are applicable:

1. **When the program must parse an algebraic string.** This case is fairly obvious. The program is asked to carry out its operations based on a computation where the user enters an equation of some sort. This frequently occurs in mathematical-graphics programs, where the program

renders a curve or surface based on any equation it can evaluate. Programs like *Mathematica* and graph drawing packages such as *Origin* work in this way.

2. **When the program must produce varying kinds of output.** This case is a little less obvious, but far more useful. Consider a program that can display columns of data in any order and sort them in various ways. These programs are frequently referred to as Report Generators, and while the underlying data may be stored in a relational database, the user interface to the report program is usually much simpler then the SQL language which the database uses. In fact, in some cases, the simple report language may be interpreted by the report program and translated into SQL.

Sample Code

Let's consider a simplified report generator that can operate on 5 columns of data in a table and return various reports on these data. Suppose we have the following sort of results from a swimming competition:

```
Amanda McCarthy            12  WCA         29.28
Jamie Falco                12  HNHS        29.80
Meaghan O'Donnell          12  EDST        30.00
Greer Gibbs                12  CDEV        30.04
Rhiannon Jeffrey           11  WYW         30.04
Sophie Connolly            12  WAC         30.05
Dana Helyer                12  ARAC        30.18
```

where the 5 columns are *frname, lname, age, club* and *time.* If we consider the complete race results of 51 swimmers, we realize that it might be convenient to sort these results by club, by last name or by age. Since there are a number of useful reports we could produce from these data in which the order of the columns changes as well as the sorting, a language is one useful way to handle these reports.

We'll define a very simple non-recursive grammar of the sort

```
Print lname frname club time sortby club thenby time
```

For the purposes of this example, we define the 3 verbs shown above:

```
Print
Sortby
Thenby
```

and the 5 column names we listed earlier:

```
Frname
Lname
Age
Club
Time
```

For convenience, we’ll assume that the language is case insensitive. We’ll also note that the simple grammar of this language is punctuation free, and amounts in brief to

Print var[var] [sortby var [thenby var]]

Finally, there is only one main verb and while each statement is a declaration, there is no assignment statement or computational ability in this grammar.

Interpreting the Language

Interpreting the language takes place in three steps

1. Parsing the language symbols into tokens.
2. Reducing the tokens into actions.
3. Executing the actions.

We parse the language into tokens by simply scanning each statement with a StringTokenizer and then substituting a number for each word. Usually parsers push each parsed token onto a *stack* -- we will use that technique here. We implement the Stack class using a Vector, where we have *push, pop, top* and *nextTop* methods to examine and manipulate the stack contents.

After parsing, our stack could look like this:

Type	Token	
Var	Time	<-top of stack
Verb	Thenby	
Var	Club	
Verb	Sortby	
Var	Time	
Var	Club	

Var	Frname
verb	Lname

However, we quickly realize that the “verb” *thenby* has no real meaning other than clarification, and it is more likely that we’d parse the tokens and skip the *thenby* word altogether. Our initial stack then, looks like this

```
Time
Club
Sortby
Time
Club
Frname
Lname
Print
```

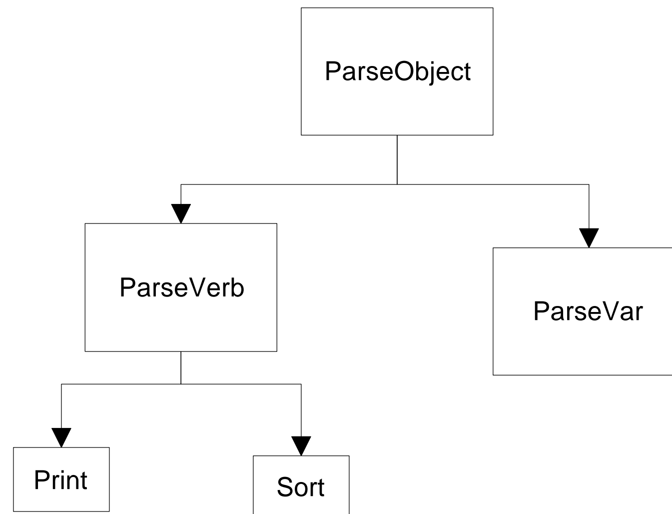
Objects Used in Parsing

Actually, we do not push just a numeric token onto the stack, but a *ParseObject* which has the both a type and a value property:

```
public class ParseObject
{
 public static final int VERB=1000, VAR = 1010,
                          MULTVAR = 1020;
   protected int value;
   protected int type;

   public int getValue() {return value;}
   public int getType() {return type;}
}
```

These objects can take on the type VERB or VAR. Then we extend this object into ParseVerb and ParseVar objects, whose value fields can take on PRINT or SORT for ParseVerb and FRNAME, LNAME, etc. for ParseVar. For later use in reducing the parse list, we then derive *Print* and *Sort* objects from ParseVerb.

This gives us a simple hierachy:



The parsing process is just the following simple code, using the StringTokenizer and the parse objects:

```
public Parser(String line)    {
      stk = new Stack();
      actionList = new Vector();

      StringTokenizer tok = new StringTokenizer(line);
      while(tok.hasMoreElements())       {
         ParseObject token = tokenize(tok.nextToken());
         if(token != null)
            stk.push(token);
      }
   }
//------------------------------------
private ParseObject tokenize(String s)       {
         ParseObject obj = getVerb(s);
         if (obj == null)
            obj = getVar(s);
         return obj;
}
//----------------------------------------
 private ParseVerb getVerb(String s)  {
         ParseVerb v;
  v = new ParseVerb(s);
  if (v.isLegal())
      return v.getVerb(s);
 else
     return null;
}
```

```
//---------------------------------------
private ParseVar getVar(String s)       {
         ParseVar v;
 v = new ParseVar(s);
 if (v.isLegal())
     return v;
 else
     return null;
}
```

The ParseVerb and ParseVar classes return objects with *isLegal* set to true if they recognize the word.

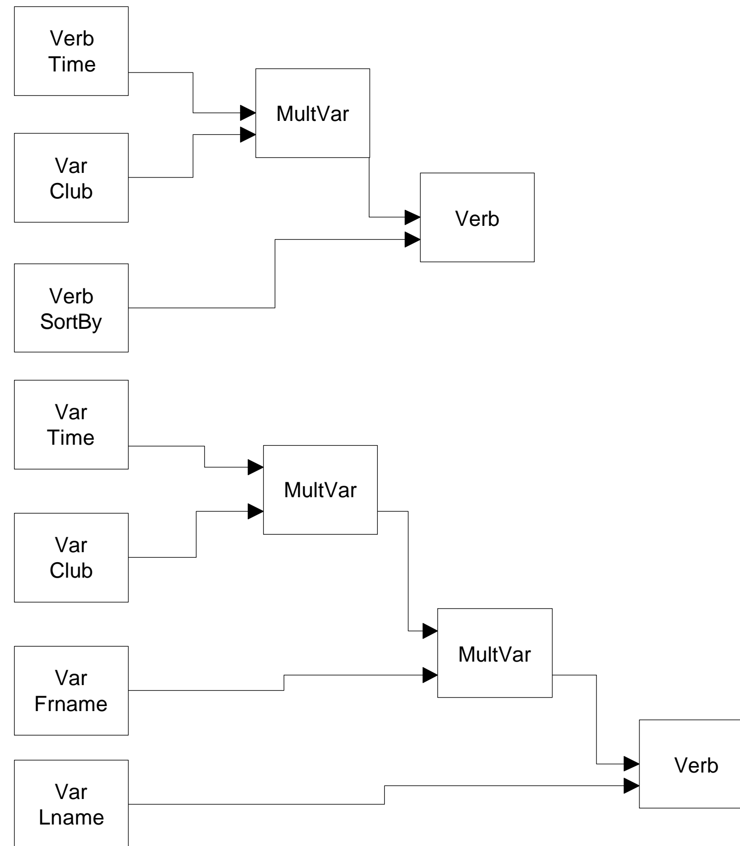
```
public class ParseVerb extends ParseObject
{
   static public final int PRINT=100,
              SORTBY=110, THENBY=120;
   protected Vector args;

public ParseVerb(String s) {
   args = new Vector();
   s = s.toLowerCase();
   value = -1;
   type = VERB;
   if (s.equals("print")) value = PRINT;
   if (s.equals("sortby")) value = SORTBY;
}
```

Reducing the Parsed Stack

The tokens on the stack have the form

```
Var
Var
Verb
Var
Var
Var
Var
Verb
```

We reduce the stack a token at a time, folding successive Vars into a MultVar class until the arguments are folded into the verb objects.



When the stack reduces to a verb, this verb and its arguments are placed in an action list; when the stack is empty the actions are executed.

This entire process is carried out by creating a Parser class that is a Command object, and executing it when the Go button is pressed on the user interface:

```
public void actionPerformed(ActionEvent e)
  {
     Parser p =  new Parser(tx.getText());
     p.setData(kdata, ptable);
     p.Execute();
  }
```

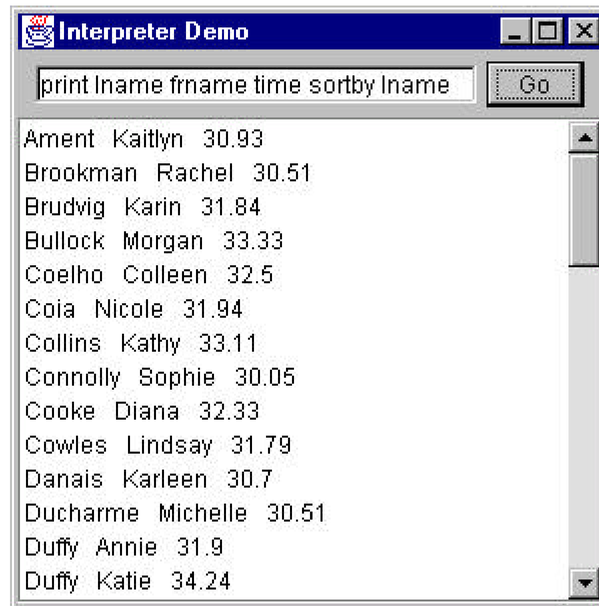
The parser itself just reduces the tokens as we show above. It checks for various pairs of tokens on the stack and reduces each pair to a single one for each of five different cases.

```
//executes parse of command line
   public void Execute()    {
    while(stk.hasMoreElements())   {
     if(topStack(ParseObject.VAR, ParseObject.VAR))
      {
       //reduce (Var Var) to Multvar
       ParseVar v = (ParseVar)stk.pop();
       ParseVar v1 = (ParseVar)stk.pop();
       MultVar mv = new MultVar(v1, v);
       stk.push(mv);
       }
     //reduce MULTVAR VAR to MULTVAR
    if(topStack(ParseObject.MULTVAR, ParseObject.VAR))
      {
       MultVar mv =  new MultVar();
       MultVar mvo = (MultVar)stk.pop();
       ParseVar v = (ParseVar)stk.pop();
       mv.add(v);
       Vector mvec = mvo.getVector();
       for (int i = 0; i< mvec.size(); i++)
          mv.add((ParseVar)mvec.elementAt(i));
       stk.push(mv);
      }
    if(topStack(ParseObject.VAR, ParseObject.MULTVAR))
      {
     //reduce (Multvar Var) to Multvar
       ParseVar v = (ParseVar)stk.pop();
       MultVar mv = (MultVar)stk.pop();
       mv.add(v);
       stk.push(mv);
       }
     //reduce Verb Var to Verb containing vars
     if (topStack(ParseObject.VAR, ParseObject.VERB))
        {
        addArgsToVerb();
        }
     //reduce Verb MultVar to Verb containing vars
     if (topStack(ParseObject.MULTVAR, ParseObject.VERB))
        {
        addArgsToVerb();
        }
     //move top verb to action list
     if(stk.top().getType() == ParseObject.VERB)
     {
        actionList.addElement(stk.pop());
     }

    }//while
    //now execute the verbs
    for (int i =0; i< actionList.size() ; i++)   {
      Verb v = (Verb)actionList.elementAt(i);
```

```
        v.Execute();
    }
  }
```

We also make the Print and Sort verb classes Command objects and Execute them one by one as the action list is enumerated.

The final visual program is shown below:



Consequences of the Interpreter Pattern

Whenever you introduce an interpreter into a program, you need to provide a simple way for the program user to enter commands in that language. It can be as simple as the Macro record button we noted earlier, or it can be an editable text field like the one in the program above.

However, introducing a language and its accompanying grammar also requires fairly extensive error checking for misspelled terms or misplaced grammatical elements. This can easily consume a great deal of programming effort unless some template code is available for implementing this checking. Further, effective methods for notifying the users of these errors are not easy to design and implement.

In the Interpreter example above, the only error handling is that keywords that are not recognized are not converted to ParseObjects and pushed onto the stack. Thus, nothing will happen, because the resulting stack

sequence probably cannot be parsed successfully, or if it can, the item represented by the misspelled keyword will not be included.

You can also consider generating a language automatically from a user interface of radio and command buttons and list boxes. While it may seem that having such an interface obviates the necessity for a language at all, the same requirements of sequence and computation still apply. When you have to have a way to specify the order of sequential operations, a language is a good way to do so, even if the language is generated from the user interface.

The Interpreter pattern has the advantage that you can extend or revise the grammar fairly easily one you have built the general parsing and reduction tools. You can also add new verbs or variables quite easily once the foundation is constructed.

In the simple parsing scheme we show in the Parser class above, there are only 6 cases to consider, and they are shown as a series of simple *if* statements. If you have many more than that, *Design Patterns* suggests that you create a class for each one of them. This again makes language extension easier, but has the disadvantage of proliferating lots of similar little classes.

Finally, as the syntax of the grammar becomes more complex, you run the risk of creating a hard to maintain program.

While interpreters are not all that common in solving general programming problems, the Iterator pattern we take up next is one of the most common ones you’ll be using.