CMSC 433 – Programming Language Technologies and Paradigms Spring 2007

Interpreter Pattern

Apr. 12, 2007

Example

- Let's consider a simplified report generator that can operate on 5 columns of data in a table and return various reports on the data.
- Suppose we have the following 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.*

Need for Reports

- 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.

The Language

- We'll define a very simple non-recursive grammar of the sort
  - e.g., Print Lname Frname Club Time SortBy Club ThenBy Time
- For the purposes of this example, we define the 3 verbs:
  - Print, Sortby, Thenby
- and the 5 column names:
  - 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; there is no assignment statement or computational ability in this language.

The Strategy

- Interpreting the language takes place in three steps:
  - Parsing the language symbols into tokens.
  - Reducing the tokens into actions.
  - 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.

The Stack

- After parsing, our stack could look like this:

Top of Stack >>

Type	Token
Var	Time
Verb	ThenBy
Var	Club
Verb	SortBy
Var	Time
Var	Club
Var	Frname
Verb	Lname
Verb	Print

Eliminating *thenby*

- We 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:

Top of Stack >>

Type	Token
Var	Time
Var	Club
Verb	SortBy
Var	Time
Var	Club
Var	Frname
Var	Lname
Verb	Print

Objects Used for Parsing

- We do not push just a numeric token onto the stack, but a *ParseObject* which has the both a type and a value property.

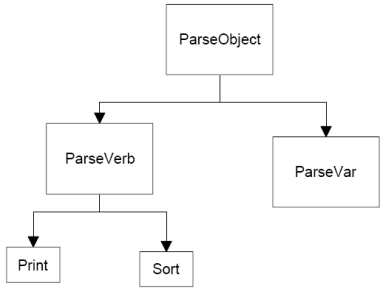
```
public class ParseObject
{
 public static final int VERB=1000, VAR = 1010,
                          MULTVAR = 1020;
   protected int value;
   protected int type;

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

- These objects can take on the type VERB or VAR.

Object Hierarchy

- 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.



Lets Start Parsing!

- 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);
      }
   }
```


tokenize()

```
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;
}
```


getVar()

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

ParseVerb

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

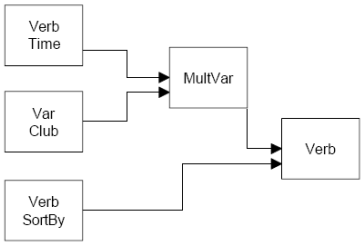
```
public class ParseVerb extends ParseObject
{
   static public final int PRINT=100,
              SORTBY=110, THENBY=120;
   protected Vector args;

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 Stack

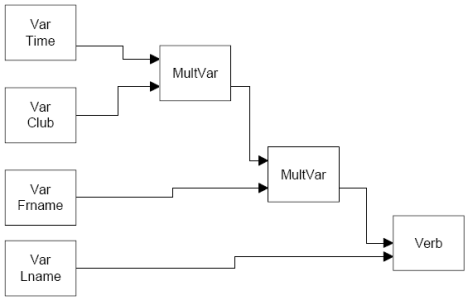
- The tokens on the stack have the form
  - Var
  - Var
  - Verb
  - Var
  - Var
  - Var
  - Var
  - Verb
- We reduce the stack a token at a time, folding successive Vars into a MultVar class until the arguments are folded into the verb objects.

The Reduction



- 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.

Reduction (contd…)

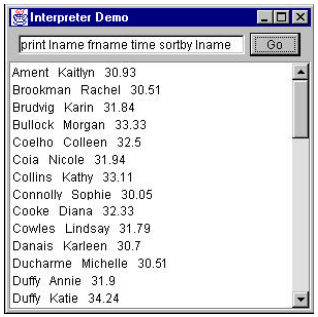


The UI

- 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. It checks for various pairs of tokens on the stack and reduces each pair to a single one for each of five different cases.



Reduce (Var Var) to MultVar

```
//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

```
 //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);
  }
```


Reduce (Var MultVar) to MultVar

```
if(topStack(ParseObject.VAR, ParseObject.MULTVAR))
  {

   ParseVar v = (ParseVar)stk.pop();
   MultVar mv = (MultVar)stk.pop();
   mv.add(v);
   stk.push(mv);
   }
```

Reducing (Verb Var) and (Verb MultVar)

```
//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();
   }
```


Preparing for Execution!

```
//move top verb to action list
if(stk.top().getType() == ParseObject.VERB)
{
   actionList.addElement(stk.pop());
}
```


Execution!

```
//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.

Concluding Remarks

- 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 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.

Concluding Remarks (contd…)

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