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:

<table>
<thead>
<tr>
<th>Name</th>
<th>Club</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amanda McCarthy</td>
<td>WCA</td>
<td>29.28</td>
</tr>
<tr>
<td>Jamie Falco</td>
<td>HNS</td>
<td>29.80</td>
</tr>
<tr>
<td>Meaghan O’Donnell</td>
<td>EDST</td>
<td>30.00</td>
</tr>
<tr>
<td>Greer Gibbs</td>
<td>CDK</td>
<td>30.04</td>
</tr>
<tr>
<td>Shannon Jeffrey</td>
<td>WYW</td>
<td>30.04</td>
</tr>
<tr>
<td>Sophie Connolly</td>
<td>WAC</td>
<td>30.05</td>
</tr>
<tr>
<td>Dana Belyer</td>
<td>APAC</td>
<td>30.18</td>
</tr>
</tbody>
</table>

where the 5 columns are fname, lname, age, club and time.

The Language

- We’ll define a very simple non-recursive grammar of the sort
  - e.g., \texttt{Print Lname Fname Club Time SortBy Club ThenBy Time}
- For the purposes of this example, we define the 3 verbs:
  - \texttt{Print}, \texttt{Sortby}, \texttt{Thenby}
- and the 5 column names:
  - \texttt{Fname}, \texttt{Lname}, \texttt{Age}, \texttt{Club}, \texttt{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
  - \texttt{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.

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.

Interpreter Pattern

Apr. 12, 2007
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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>Time</td>
</tr>
<tr>
<td>Verb</td>
<td>ThenBy</td>
</tr>
<tr>
<td>Var</td>
<td>Club</td>
</tr>
<tr>
<td>Verb</td>
<td>SortBy</td>
</tr>
<tr>
<td>Var</td>
<td>Time</td>
</tr>
<tr>
<td>Var</td>
<td>Club</td>
</tr>
<tr>
<td>Var</td>
<td>Fname</td>
</tr>
<tr>
<td>Verb</td>
<td>Lname</td>
</tr>
<tr>
<td>Verb</td>
<td>Print</td>
</tr>
</tbody>
</table>

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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Token</th>
</tr>
</thead>
<tbody>
<tr>
<td>Var</td>
<td>Time</td>
</tr>
<tr>
<td>Var</td>
<td>Club</td>
</tr>
<tr>
<td>Verb</td>
<td>SortBy</td>
</tr>
<tr>
<td>Var</td>
<td>Time</td>
</tr>
<tr>
<td>Var</td>
<td>Club</td>
</tr>
<tr>
<td>Var</td>
<td>Fname</td>
</tr>
<tr>
<td>Var</td>
<td>Lname</td>
</tr>
<tr>
<td>Verb</td>
<td>Print</td>
</tr>
</tbody>
</table>

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.

  ```java
  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.

```
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 parsing process is just the following simple code, using the StringTokenizer and the parse objects.

```
public Parse(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);
    }
}
```

```
Let's Start Parsing!

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

```
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 \textit{isLegal} set to true if they recognize the word.

```java
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;
    }
}
```

The tokens on the stack have the form
- \textit{Var}
- \textit{Var–Verb–Var}
- \textit{Verb}

We reduce the stack a token at a time, folding successive \textit{Vars} into a \textit{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…)

Reducing the Stack (contd…)

- The tokens on the stack have the form
- \textit{Var}
- \textit{Var–Verb–Var–Var–Verb}
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.

- 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

```java
//executes parse of command line
public void Execute() {
    while(stk.hasMoreElements()) {
        if(topStack(ParseObject.VAR, ParseObject.VAR))
        {
            //reduce (Var Var) to MultVar
            ParseVar v1 = (ParseVar)stk.pop();
            ParseVar v2 = (ParseVar)stk.pop();
            MultVar mv = new MultVar(v1, v2);
            stk.push(mv);
        }
    }
}
```

Reduce (MultVar Var) to MultVar

```java
//reduce MULTVAR VAR to MULTVAR
if(topStack(ParseObject.MULTVAR, ParseObject.VAR))
{
    MultVar mv = new MultVar();
    MultVar mv0 = (MultVar)stk.pop();
    ParseVar v = (ParseVar)stk.pop();
    mv.add(v);
    Vector mvec = mv0.getVector();
    for (int i = 0; i < mvec.size(); i++)
    {
        mv.add((ParseVar)mvec.elementAt(i));
    }
    stk.push(mv);
}
```

Reduce (Var MultVar) to MultVar

```java
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)

```java
//reduce Verb Var to Verb containing vars
if (topStack(ParseObject.VAR, ParseObject.VERB))
    addArgToVerb();
//reduce Verb MultVar to Verb containing vars
if (topStack(ParseObject.MULTVAR, ParseObject.VERB))
    addArgToVerb();
```

Preparing for Execution!

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

Execution!

```java
//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.