Abstract

Oops3, targeted to Java and C#, is the latest in a family of LL(1) parser generators based on an object-oriented architecture that carries through to the parsers it generates. Moreover, because oops3 employs several design patterns, its rich variety of APIs are nevertheless easy to learn. This paper discusses the use of patterns in the oops3 system and the resulting benefits for users and educators.

1. Introduction

A parser checks a program written in a source language for syntactic correctness and arranges for further processing by other means using some host language. A parser generator usually accepts an annotated grammar of the source language and, as a minimum, produces the recognition part of the parser. Because an annotated grammar is just one more source language, a parser generator is a specific case of a parser and can be used to bootstrap its own implementation.

A widely used example is Java's API for XML Parsing (JAXP) [1] with the abstract parser generators SAXParserFactory and DocumentBuilderFactory and the parsers SAXParser and DocumentBuilder. The names hint at the use of the Factory design pattern. The two factories can be configured to deliver any parser implementations as long as they are derived from the abstract parser classes. Moreover, SAXParser uses Observer patterns to report on various aspects of recognition and DocumentBuilder returns a Document which is both a container and factory for the nodes in the tree representing an XML document. Unfortunately, the consequent use of design patterns falls short: Document cannot be overridden prior to its use by a DocumentBuilder.

In the oops3 system [2] the parser generator represents a grammar of a source language as a serialized tree. The Visitor pattern is used to implement various tree manipulations, among them recognition of the source language. Template methods allow the recognition visitor to be subclassed to support various ways to observe recognition. By far the most convenient subclass is Build where the recognizing visitor uses reflection on rule names to send messages with collected tokens when grammar rules are reduced.

```java
public class parser {
  
  <Integer>  Number = '[0-9]+';
  <>         sum: term (add | subtract)*;
  <Add>      add: '+' term;
  <Subtract> subtract: '-' term;
  term: Number | '(' sum ')';
  
  
}
```
Based on source grammar annotations oops3 can generate a factory and classes to represent the source program; the factory acts as an observer to Build. As an example, the input shown in figure 1 is all that is required for oops3 to generate a parser that will recognize expressions of sums and differences and represent them as left-associative trees of Add and Subtract nodes with Integer leaves as shown in figure 2.

```
Add
  Add
  Integer 1
Subtract
  Integer 2
  Integer 3
  Integer 4
```

Figure 2: Tree for 1+(2-3)+4.

2. Parser Factory

The parser generator accepts an annotated grammar specified in one of several extended BNF notations and represents it as a tree using the following classes:

- A Parser is a container for rules.
- A Rule connects a nonterminal name to an explanation built from nodes.
- A Literal node describes a self-explaining terminal symbol.
- A Token describes terminal symbols such as numbers which are explained with patterns in the annotated grammar and require additional information obtained from the scanner.¹
- A Nonterminal references a rule.
- Other nodes are containers: Sequence, Xor for exclusive alternatives, and Repeat for iterations.

While these classes suffice to represent typical extended BNF notations, oops3 provides several other notations and associated classes to further simplify expression of grammars. And and Or, similar to Xor, represent complete and partial permutations; Delimit, similar to Repeat, represents delimited iterations; and AndList and OrList represent delimited permutations.

oops3 uses the Visitor pattern for all tree manipulations. Subclass relationships among the tree classes can still be exploited. A visitor method delegates to another one in the following fashion:

```java
Object visit (And node) { return visit((Xor)node); }
```

Visitor is the abstract base class of all visitors and contains this kind of code; therefore, a specific visitor only implements those visit methods which it does not choose to inherit.

Notations for extended BNF can describe themselves, i.e., the parser generator can be

used to turn its own input language into a tree represented by Parser and the other classes. As we shall see in the next section, all algorithms in oops3 are implemented as visitors operating on this kind of a tree; therefore, the system is bootstrapped by manually creating a tree to represent one notation for extended BNF. To keep things as flexible as possible, Parser and the other classes are hidden behind a ParserFactory.

![Figure 3: Compiling an arithmetic expression.](image)

In Figure 3, the ParserFactory on the left is used by the bootstrap code Boot to build a representation for the grammar for an extended BNF notation as a tree. The tree is serialized and stored in the file boot.ser. Parser uses the tree stored in boot.ser and an extended BNF grammar for arithmetic expressions stored in the text file expr.rfc to set up the data structures that will be used to recognize arithmetic expressions. Through Build, the observer RfcBuilder, and ParserFactory (at bottom), the expression grammar is built as a tree named expr.ser (top middle) which in turn can be used by Build (bottom right) to recognize an arithmetic expression expr.txt (at right) and represent it as a tree (right top) using expression-specific classes in Expr.java which are generated by oops3 from annotations in the grammar.

3. Visitors

The essential algorithm for a parser is recognition. The next section describes how arrangements are made so that the user of a parser can interact with the recognition process. A parser generator needs additional algorithms to generate a parser and to ensure that generated parsers operate in a predictable manner. As discussed in the previous section, oops3 represents a grammar as a tree over the Parser classes. Therefore, all algorithms, meaning in our case visitor subclasses, will operate on these classes.

- **Lookahead** attributes each node in a grammar tree with the set of input symbols that determines if recognition is possible.
- **Recognize** uses the sets produced by Lookahead and attempts the recognition of a source program.
- **Recursive** detects unlimited recursion in a grammar specification.
- **Follow** computes for each node a set of input symbols which can follow the source
recognized by the node.

- **LL1** combines the results of **Lookahead** and **Follow** to decide if a grammar is unambiguous and suitable for recursive descent parsing as implemented by **Recognize**.

- **Gen** can generate a Java program which may contain a main program, a scanner, and a tree factory to flesh out the recognition process.

In an earlier paper [5] we discussed all the algorithms but **Gen** in detail and argued that introducing the **Parser** classes has the benefit of a divide-and-conquer approach which can be used in a classroom to actually let students discover the algorithms.

In **oops3**, unlike in its predecessors, the Visitor pattern is applied to implement each node-processing algorithm. The expected benefit is to allow presentation of an algorithm within a single class, in spite of the fact that the algorithm has been divided into actions that are specific to about a dozen classes — very similar to what is expected from aspect-oriented programming (AOP) [6].

Specifically for recognition the Visitor pattern results in an additional quite unexpected benefit. To be useful, recognition has to be combined with some observation mechanism so that the user of a parser can interact with the progress of recognition. As is discussed in the next section, the Visitor pattern in combination with subclassing allows the implementation of two very different observation protocols. Even more could potentially be added. While this was approximated in an earlier version of **oops** [7] by subclassing the **Parser** classes, the Visitor pattern has the essential advantage that a production parser can be delivered with just those visitors that are required for a particular compiler application. Just like aspects the visitors are separated well enough to be included selectively.

Figure 3 suggests that only **Lookahead** and **Recognize** (in form of a subclass such as **Build**) are involved in parsing. This is indeed the case: **oops3** is operational even if the other visitors are omitted. In fact, this configuration, made possible by the Visitor pattern, allows for a simple bootstrap of the parser generator, and at the same time makes a strong case for the judicious use of ambiguous grammars. **Lookahead** checks that alternatives in **Xor** and similar nodes are selected in a deterministic fashion. **Follow** and **LL1** are only required to check if there is overlap between an iteration in **Repeat** and whatever follows a **Repeat** node. Because **Recognize** implements a greedy behavior for **Repeat** and related classes, it functions even for an ambiguous grammar by collecting the longest possible input sequence — the time-honored approach to the 'dangling else' problem.

### 4. Observers

Figure 4 summarizes the essential aspects of the recognition algorithm implemented in **Recognize**. To avoid backtracking, a node is only visited if the current input symbol matches the set computed by **Lookahead**; therefore, visits to a **Literal** or **Token** node need only advance the input, and **Xor** can select the proper descendant by requiring that all descendants’ lookahead sets must be disjoint.

The numbers in figure 4 mark the methods of **Recognize** which are particularly interesting for observing the recognition process. For example, if visits to **Literal** and **Token** simply add the current input to a global list, the list will eventually contain
the symbols of the source program. A syntax tree results if each visit to a Rule sets up its own list and Nonterminal arranges for nesting and labeling with the nonterminal name.

<table>
<thead>
<tr>
<th>Tree node</th>
<th>Example</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rule</td>
<td>&lt;Add&gt; add: '+' term;</td>
<td>visit right-hand side</td>
</tr>
<tr>
<td>Literal</td>
<td>'+'</td>
<td>advance input</td>
</tr>
<tr>
<td>Token</td>
<td>Number</td>
<td>advance input</td>
</tr>
<tr>
<td>Nonterminal</td>
<td>add</td>
<td>visit rule</td>
</tr>
<tr>
<td>Sequence</td>
<td>'+' term</td>
<td>visit descendants in order</td>
</tr>
<tr>
<td>Repeat</td>
<td>( _ )*</td>
<td>greedy: visit descendant</td>
</tr>
<tr>
<td>Xor</td>
<td>add</td>
<td>subtract</td>
</tr>
</tbody>
</table>

Figure 4: Recognition visitor.

Figures 1 and 2 illustrate that better control is needed to construct useful trees. Build is a subclass of Recognize which extends three of the four methods marked in figure 4 and implements an Observer pattern similar to the reduce actions which can be programmed in the yacc parser generator and its descendants [8]: A visit to Rule sets up a list, and a visit to Token copies information about the symbol from the scanner to the current list. Once the Rule is complete, the list is offered to an observer method by reflection on the nonterminal name. Finally, a visit to a Nonterminal will add the result of the method (or the list, if reflection failed) to the current (outer) list. Plain, nested lists are flattened by convention; therefore, the observer has precise control over the shape of the resulting tree — it can even wrap several results into a list and have the list flattened as it is added by Nonterminal to the outer list. If Gen creates a tree factory as an observer to Build, it generates container classes and factory methods exactly for those rules which are annotated with class names, as shown in Figure 1.

LL(1)-based recognition cannot deal with left recursion in a grammar; therefore, left-associative operators require some ingenuity in tree building, but even that can be automated. In figure 1 the rule for sum is annotated with empty brackets indicating to Build that the collected objects should be collated into a left-associative tree. The result is the typical arithmetic tree as shown in figure 2.

Build implements an observer pattern for reduction and Gen provides a tree factory as an observer, optionally with the infrastructure for one of several visitor patterns for the tree representing the source program. Together they go a long way towards automating the representation of a source program with very fine control over the shape of the tree. The Factory pattern for the tree classes additionally makes it very simple to extend some or all of the generated classes.

Other observer patterns can be implemented. Observe is a subclass of Recognize which implements an observer interface similar to the one used in the predecessor of oops3 [9]: A visit to a Rule first sends an init message to the current observer which must respond with an observer for the rule activation — this allows context to be passed into a rule activation. Visits to Literal, Token, and Nonterminal result in shift messages. Finally, the visit to a Rule ends with a reduce message to the rule activation's observer.

The interface implemented by Observe is very similar to the SAX handlers in JAXP, but with one rather important difference: The SAX handlers are flat — a single handler receives messages about all nested XML elements. An observer for Observe can be
implemented to be flat — `init` would simply return this — or it can be implemented to be rule-specific or rule-activation-specific, thus eliminating the need for more explicit tracking of nested structures [10].

### 5. Template Methods

`Recognize` is the building block for all APIs to process the source language. Observer patterns such as `Build` and `Observe` should be implemented as subclasses of `Recognize` so that the somewhat delicate recognition algorithm is inherited and not compromised.

In general, it is not sufficient to rely on overriding and access to base class methods to support something like arbitrary `Token` collection efforts. For example, in one of the extended BNF notations supported by `oops3`, the rule

```plaintext
any: { a | b | c };
```

specifies that a, b, and c must all be represented in the input, but that they can appear in any order. In the `Parser` tree, this is represented with an `And` node. When `Recognize` visits an `And` node it ensures that all descendants are accounted for. However, further processing of a source program is likely to be simplified if `Build` represents a, b, and c in the source tree in the order in which they are specified in the grammar, rather then in the order in which they appear in a particular source program.

This means that `Build` needs to arrange for separate collection lists for a, b, and c. That is, while visiting an `And` node `Recognize` has to use a template method to actually recognize a descendant so that `Build` can override the template method and arrange for separate collection.

Permutations are only one example for the need for template methods. To be completely flexible, one can argue that any processing of a descendant should involve a template method, e.g., to trace a particular iteration or sequential processing, to sort permuted input, to postprocess delimiters, etc. If one considers observation an aspect of recognition, implemented by subclassing, the architecture of `Recognize` simply has to provide access to all interesting events by means of template methods. At this point `oops3` only tries to strike a happy medium between full flexibility and an overwhelming number of template methods.

### 6. Summary

The paper discussed `oops3` as an example for the consequent use of design patterns in parsing and parser generation and it pointed out significant benefits of the architecture.

The central concept is to represent source programs as trees and to implement tree manipulation using the Visitor pattern. Tree classes usually are specific to the source grammar and provide natural boundaries for divide-and-conquer in all algorithms. The Visitor pattern combines the class-specific pieces of an algorithm in a central class.

Recognition is implemented as a visitor with template methods. It is subclassed to provide different ways to observe the recognition process. One particular Observer pattern instance connects recognition to a tree factory to represent a source program; the tree factory can be generated from annotations in the source grammar. The Factory pattern makes it simple to extend the tree classes.
The parser generator itself is a special case of a parser and is used to implement itself. It uses the same classes as any other parser and is bootstrapped from a hand-crafted series of calls on the factory. Because of self-compilation the parser generator can support different notations for extended BNF, including new extensions to deal with permutations and delimited lists.

The consequent use of design patterns results in a very modular system with well-separated algorithm implementations and with very flexible parsing APIs to interact with the recognition process while maintaining a very flat learning curve for typical applications.

References