Design Patterns for Parsing

Dung ("Zung") Nguyen and Stephen B. Wong
Dept. of Computer Science
Rice University
Houston, TX 77005
dxnguyen@rice.edu, swong@rice.edu

Abstract

We provide a systematic transformation of an LL(1) grammar to an object model that consists of:

∉ an object structure representing the non-terminal symbols and their corresponding grammar production rules,

∉ a union of classes representing the terminal symbols (tokens).

We apply the visitor pattern to the above union of token classes to model a predictive recursive descent parser on the given grammar. Parsing a non-terminal is represented by a visitor to the tokens. The abstract factory pattern, where each concrete factory corresponds to a non-terminal symbol, is used to manufacture appropriate parsing visitors.

Our object-oriented formulation for predictive recursive descent parsing eliminates the traditional construction of the predictive parsing table and yields a parser that is declarative and devoid of conditionals. It not only serves to teach not only standard techniques in parsing but also as a non-trivial exercise of object modeling for object-first introductory courses.

We implement our design in Java and make extensive applications of the powerful concept of closure via anonymous inner classes.

1 Introduction

The 2001 ACM Computing Curricula lists the object-first approach as a legitimate way to teach object-oriented programming (OOP) in introductory computer science courses [1]. OOP educators would concur that in order for such courses to be effective, they must progress normally yet quickly to cover topics that are complex enough to make a compelling case for OOP (see for instance, [2][3]). A wealth of problems in various phases of a compiler can be appropriately modeled as object-oriented systems. However, such problems are rarely discussed at the introductory level in current computer science curricula.

A quick tour of web sites and extant textbooks [4][5][6][7] seems to indicate that context-free grammars (CFG) and their related topics are usually relegated to upper division courses in programming languages and compiler construction. Efforts have been made to introduce object-oriented design patterns such as the composite and visitor patterns into such courses at the semantic analysis phases but not at the syntax analysis phase [5][8]. Perhaps because it is considered well understood, the current treatment of predictive recursive descent parsing (PRDP), typified by the construction of a predictive parsing table and the use of a large stack of conditionals on the token type to select the appropriate production rule, offers no innovation and incorporates no object-oriented concepts. Such a procedural approach does not scale, is rigid and cannot easily adapt to change: a small modification in the grammar such as adding a new production rule for an existing non terminal symbol will require a complete rewrite of the code.

We present in this paper an object-oriented formulation of PRDP for LL(1) grammars that is flexible and extensible, yet simple enough to be taught in a CS2 object-first course. The key design element is to equip the tokens with the capability to perform an open-ended number of tasks and to shift the responsibility of determining what production rule to parse to the tokens themselves. This calls for the visitor pattern [9]. The details of how it used to model the terminal symbols (tokens) of a grammar are described in Section 2.

We also need to model the non-terminal symbols of a grammar and their corresponding production rules, which define the form (syntax) for all sentences generated by the given grammar. Section 3 illustrates via a simple example, a systematic transformation of a given LL(1) grammar to an equivalent grammar where each non terminal symbol translates to a class/interface whose production rules are expressed in terms of "has-a" and "is-a" relationships. The composite pattern [9] is used extensively here, resulting in

an object model that represents the meta-structure of the parse tree.

With the object model for the complete grammar in place, section 4 shows how the tokens parse by "accepting" concrete visitors, each of which corresponds to a leaf class in the parse tree structure or a union of other such visitors. This approach lacks modularity and requires looking deep into the parse tree structure thus breaking its encapsulation. The result is tightly coupled visitor code that leaves no room for changes in the grammar.

To decouple the parsing visitors and achieve a much higher level of modularity, we apply the abstract factory pattern [9] and relegate the manufacturing of parsing visitors to appropriate concrete factories instead. Section 5 explains how such a design helps produce a robust object-oriented predictive recursive descent parser that requires only local knowledge of each grammar rule and as a result is flexible and readily extensible.

2 Visitor Pattern for Tokens

We represent the abstract notion of a token with an abstract class *AToken*. For our purpose, the only attribute we are interested in is its string representation called lexeme. Each token in a given grammar is simply a concrete subclass of *AToken*. *AToken* and a collection of concrete subclasses constitute what we called a "union" of token classes. A tokenizer to extract tokens from an input stream is represented by an interface called *ITokenizer*.

Parsing a finite sequence of tokens means asking the tokens to determine whether or not their linear structure conform to the syntax rules defined in a given grammar. Since the same set of tokens can be used in different grammars, tokens cannot have any knowledge about any grammar nor any action one wants to do with them. To have any "intelligence" at all, a token must then be equipped with a hook for an open-ended extension of behaviors [10]. That's where the visitor pattern comes in: the union of tokens serves as the host and the tasks that one wants to perform with these tokens are the visitors. The visitor interface, called *ITokenVis*, will consist of a finite number of methods; each corresponds to the behavior of a concrete host token.

In the procedural approach, at any given stage of the parsing, one must externally determine if the current token is one of all the possible tokens and then decide what to do based on its type. In contrast, with the visitor pattern, we only need to write the behavior we want for each of the tokens in a concrete *ITokenVis*, ask the current token to "accept" the visitor and let polymorphism direct the control flow. In practice, we expect to encounter only a handful of tokens at any given stage and need to define specific actions for only these tokens. The rest we can ignore and

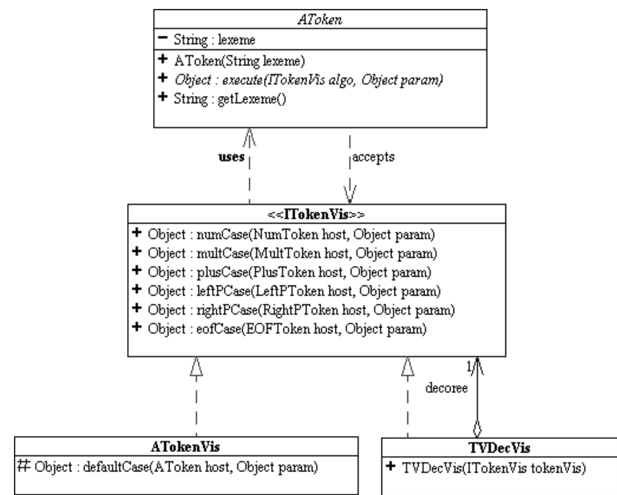

Figure 1: Class diagram of the visitors to a token.

throw an exception. Thus for convenience, all token visitors are derived from an abstract implementation of *ITokenVis*, called *ATokenVis*, whose default behavior is to throw an exception. It is also convenient in practice to add a special token called EOF to mark the end of the input string. The visitor interface thus will need to have a method for this EOF token as well.

As an example, consider the classic CFG for infix arithmetic expression:

E :: T **+** E | T
T :: **num** | **num** ***** T | **(** E **)**

The visitor interface will have a method for each of the tokens **+**, *****, **(**, **)**, **num** and the special EOF token. The UML class diagrams for the visitors and its abstract host are depicted in Figure 1.

3 Composite Structure for Non-Terminals and Production Rules

The grammar given in Section 2 is not LL(1) but can be left-factored to yield the following equivalent LL(1) grammar.

E :: T E'
E' :: *empty* | **+** E
T :: **num** T' | **(** E **)**
T' :: *empty* | ***** T

This grammar isn't quite ready to be modeled as classes however. This is because there are still sequences of symbols, such as "+ E" and "num T'", that are not yet associated with a unique symbol. So, we perform one more grammar transformation where each distinct sequence of two or more symbols on the right hand side of the production rules is given a unique non-terminal symbol on the left hand side of the rules. It is clear that this is an

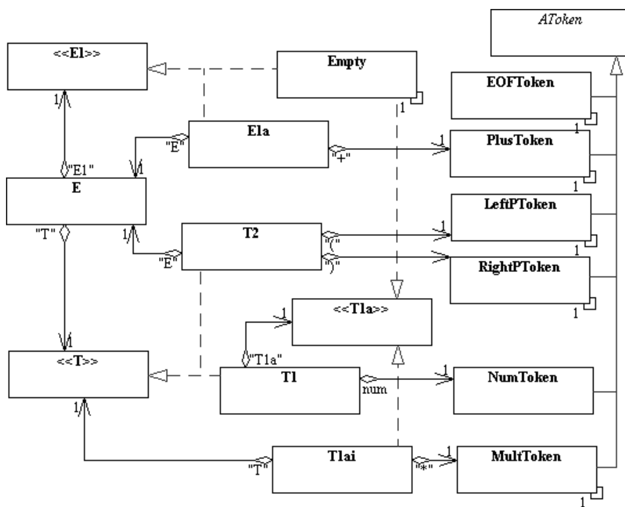

Figure 2: Object model of the example grammar

equivalent grammar because it simply gives names to existing sequences of symbols. Below, we have changed the names slightly to remove the primes and create Java-legal symbol names:

E :: T E1
E1 :: ***empty*** | E1a
E1a : **+**E
T :: T1 | T2
T1 :: **num** T1a
T1a :: ***empty*** | T1ai
T1ai :: ***** T
T2 :: **(**E**)**

In an object model, there are two fundamental types of relationships, "is-a", represented by inheritance, and "has-a", represented by composition. Thus, in order to create an object model of our grammar, we need to see if these two relations are expressed by the grammar. What we see is that non-terminals with more than one production rule ("branches") can be represented with an "is-a" inheritance relationship because the union of those production rules says that any terms on the right hand side can be represented by the left hand side. The second, fourth and sixth rules above are branches. On the other hand, some rules represent a sequence of terms, such as the first, third, fifth, seventh and eighth rules above. The left hand side of these "sequences" can be said to be composed of the right hand side terms. Thus the distinct non-terminal sequences can be represented by compositional relationships.

We can now simply and directly create our object model of the grammar. In Figure 2, we see that all the non-terminals are represented by classes or interfaces. Branches are represented by interfaces, to allow multiple inheritances, and sequences are represented by classes because they require fields. In addition, all the terminal symbols, which are the possible tokens, such as "+", "*" and numbers, are represented by their own classes. The empty term is represented by its own class as is the end-of-file token. The recursive nature of the grammar is immediately evident as the composite design pattern in the class structure.

If the above object structure is indeed a good representation of the grammar it models, then it will contain all the relationships, features and other information in that grammar. Therefore, instead of doing a large-scale case analysis over the entire grammar, if we let the object structure drive the processing of a token stream, then all the necessary case information will automatically be present.

4 Parsing with Concrete Visitors

The naïve first attempt at parsing an input token stream using visitors involves instantiating the required visitors and then executing the visitor for the start symbol on the first token.

One quickly runs into trouble however. The problem with directly defining and instantiating the parsing visitors is that at any given stage, one must analyze the details of the grammar to the level of knowing what are the possible tokens at that stage. This analysis may require one to look beyond the immediate relationships a class may have with any other classes. For instance, to find the possible tokens that could be the start of an "E" term, one must look 2 levels down to the "T1" class. In a good OO system however, one wants to encapsulate and isolate each code section from any other. The code to process an E term should only be concerned with the fact that an E term is composed of a T term and an E1 term, and not be concerned with what the internal nature of the T, E1 or any other terms.

In order to create a visitor that parses a branch, one must create the union of all the visitors that parse the branch's subclasses. If the subclasses are all sequences, we are forced to duplicate their methods, which correspond to their respective first tokens in their sequences, in the visitor for the branch. Luckily, an LL(1) grammar insures that there are conflicts between methods because each token uniquely determines a sequence. But if one of the branch's subclasses is also a branch, we must look further down to find a sequence and then create the union of all those visitors as well. This process is not unlike the traditional PRDP methodologies that utilize a global case analysis to deduce the prediction table. Thus, the direct instantiation of the parsing visitors requires that we a) know exactly how all the subclasses process their tokens and b) replicate their code in any superclasses (branches).

Direct instantiation of the parsing visitors can be done and does generate some very compact code. But in addition to the above mentioned violation of encapsulation and

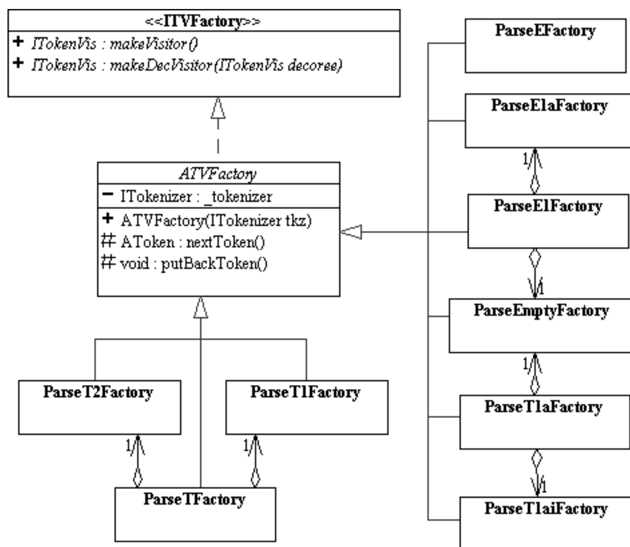

Figure 3: Class diagram of the factories of the visitors to a token.

replication of code, the non-local analysis of our naïve approach causes strong coupling in our system, which limits its ability to scale to larger, more complex grammars.

5 Factories

To remedy this problem, one must re-think the instantiation process of the visitors. The abstract factory design pattern is very useful for abstracting and encapsulating the construction of classes. Using factories to instantiate the parsing visitors

1. Enables each term to be decoupled from any other term by hiding the instantiation details.
2. Enables the construction of the union of visitors by decoration, which is used to implement branches.
3. Enables the delayed construction of already installed visitors which is needed to create circular relationships.

Each non-terminal symbol (and its corresponding class) is associated with a factory that constructs its parsing visitor (See Figure 3). All factories adhere to a basic factory interface which provides the methods to instantiate the parsing visitors. For convenience sake, all the factories are derived from an abstract factory, *ATVFactory*, that provides access to the tokenizer.

The constructors of factories for sequence terms (e.g. E, T1, E1a & T1ai -- see Listing 1) will take in the factories for their composed terms. Generally, they simply ask the supplied factory to immediately make the required visitor (using *IFactory*.makeVisitor()), which is then stored for later use. Listing 1 shows how the use of anonymous inner classes in the makeVisitor() method to instantiate the parsing visitor, creates a closure that includes the stored

```
public class ParseE1aFactory extends ATVFactory {
  private ITokenVis _parseE;

  public ParseE1aFactory(ITokenizer tkz, ParseEFactory eFac){
     super(tkz);
     _parseE = eFac.makeVisitor(); }

  ;; the following constructor and settor used for circular relations
  public ParseE1aFactory(ITokenizer tkz){  super(tkz); }

  public void setParseEFactory(ParseEFactory eFac) {
    _parseE = eFac.makeVisitor(); }

  private E1a makeE1a(PlusToken host, Object inp) {
     return new E1a(host, (E)nextToken().execute(_parseE, inp)); }

   public ITokenVis makeVisitor() {
     return new ATokenVis() {
        public Object plusCase(PlusToken host, Object inp) {
           return makeE1a(host, inp);  } }; }

   public ITokenVis makeDecVisitor(ITokenVis decoree) {
     return new TVDecVis(decoree) {
        public Object plusCase(PlusToken host, Object inp) {
           return makeE1a(host, inp); } }; }
}
```

Listing 1: Typical factory for a sequence parsing visitor.

visitor, _parseE. Thus, even if the makeVisitor() or makeDecVisitor() methods are called before _parseE has been initialized, the generated visitors will run properly so long as _parseE is initialized before they are used. This is a powerful example to students of how closures enable one to control objects to which one may not even have a direct reference.

The constructors of factories for branch terms (e.g. T, E1, & T1a -- see Listing 2) take in the factories for all the branch's subclasses. In this case, the supplied factories themselves are stored. A visitor that parses a branch is the union of all the visitors that parse its subclasses. Since the grammar is LL(1), and each method of a visitor corresponds to a particular token, none of the subclasses' visitors utilize the same method for processing. Thus the union of the subclasses' visitors can be accomplished by

```
public class ParseTFactory extends ATVFactory {
   private ParseT1Factory _t1Fac;
   private ParseT2Factory _t2Fac;

   public ParseTFactory(ITokenizer tkz, ParseT1Factory t1Fac,
                                             ParseT2Factory t2Fac) {
      super(tkz);
      _t1Fac = t1Fac;
      _t2Fac = t2Fac;  }

   public ITokenVis makeVisitor()  {
      return _t1Fac.makeDecVisitor(_t2Fac.makeVisitor()); }

   public ITokenVis makeDecVisitor(ITokenVis decoree) {
      return _t1Fac.makeDecVisitor(
                     _t2Fac.makeDecVisitor(decoree)); }
}
```

Listing 2: Typical factory for a branch parsing visitor.

```
ITokenizer tok = new Tokenizer(filename);

ParseT1aiFactory t1aiFac = new ParseT1aiFactory(tok);
ParseT2Factory t2Fac = new ParseT2Factory(tok);
ParseE1aFactory e1aFac = new ParseE1aFactory(tok);

ParseTFactory tFac =
   new ParseTFactory(tok, new ParseT1Factory(tok,
      new ParseT1aFactory(tok, t1aiFac,
         new ParseEmptyFactory(tok))), t2Fac);

t1aiFac.setParseTFactory(tFac);

eFac = new ParseEFactory(tok, tFac,
   new ParseE1Factory(tok, e1aFac,
      new ParseEmptyFactory(tok)));

t2Fac.setParseEFactory(eFac);
e1aFac.setParseEFactory(eFac);

ITokenVis  parseEVisitor = eFac.makeVisitor();
```

Listing 3: Instantiation of the parsing visitor factories, where circular references exist.

using the decorator design pattern [9]. But since the factory for the branch doesn't know what methods are utilized by the subclasses' visitors, it is forced to delegate the decoration process to once of the factories of the subclasses. Hence, all factories provide a method to produce a visitor that is the decoration of a supplied visitor (*IFactory*.makeDecVisitor(IFactory decoree)). This method simply overrides the appropriate method of a base decorator class, TVDecVisitor (see Figure 1) whose methods simply default to delegating to the corresponding method of the decoree. The E and Empty terms are special cases since they have defined behaviors for all token cases. Thus these terms can only be decorated and cannot be used to decorate another visitor.

The result is that instead of constructing the parsing visitors directly, one now constructs the parsing visitor factories (See Listing 3). Each factory's construction only requires the factories of those terms it is directly related to, either by composition or by subclass. One thus need only know the grammar one level at a time, no global knowledge of the grammar is needed. This decoupling of the grammar terms makes the system very robust with respect to changes in the grammar.

6 Conclusion

We have created an object-oriented predictive recursive descent parser by starting with an LL(1) context-free grammar and applying a simple transformation. The resulting equivalent grammar was directly modeled by a class structure using inheritance to represent branches and composition to represent sequences. Since the tokens determine whether or not the input corresponds to the grammar, the visitor design pattern was used to provide direct dispatching to the appropriate parsing code, thus eliminating conditionals. The code thus became declarative in nature. The abstract factory pattern was used to decouple the individual grammar elements from each other and create a flexible, extensible system. The traditional global case analysis, predictive parsing table and attendant stack of conditionals gave way to a simple local analysis and delegation-based behavior. Decorators were used to model the union of parsing behaviors needed under branching conditions. While it is beyond the scope of this paper, the object structure of the parse tree can easily be extended with its own visitors to enable semantic analysis of the parsed input.

It is important to recognize that OO PRDP cannot be taught in isolation. It must be carefully integrated into an objects-first curriculum that emphasizes OOP/OOD, design patterns, and abstract decomposition. At our institution, this material is covered near the end of CS2, which an OO data structures and algorithms course. At this point in the curriculum, the students are already versed in basic OOP/OOD practices, including all the design patterns mentioned here. The PRDP formulation serves not only to expose the students to fundamentals of syntactic analysis, but also serves as a vehicle for teaching them how to decompose a problem into flexible and extensible object system.

References

[1] Computing Curriculum 2001, Computer Science Volume, Dec. 15, 2001 (http://turing.acm.org/sigs/sigcse/cc2001/)

[2] Madsen, Ole, Keynote speech at OOPSLA 2002, Seattle, WA, Nov. 7, 2002. (oopsla.acm.org/fp/files/spe-concepts.html)

[3] Alphonce, C., Nguyen, D., Ventura, P. and Wong, S., "Killer Examples" for Design Patterns and Objects First Workshop, OOPSLA 2002, Seattle, WA. Nov. 4, 2002. www.cse.buffalo.edu/~alphonce/OOPSLA2002/KillerExamples

[4] Aho, A., Sethi, R., and Ullman, J., *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[5] Appel, A., Palsberg, J., *Modern Compiler Implementation in Java, 2nd ed.*, Cambridge University Press, 2002.

[6] Grune, D., Bal, H., Jacobs, C., and Langendoen, K., *Modern Compiler Design*, Wiley, 2000.

[7] See for instance:

inst.eecs.berkeley.edu/~cs164/

penguin.wpi.edu:4546/course/CS544/PLT4.4.html

www.cs.cornell.edu/courses/cs211/2000fa/materials/Lecture09-Sept-26-Recursive-Descent-Parsing.pdf

www.cs.nyu.edu/courses/spring02/G22.2130-001/parsing1.ppt

http://www.cs.rit.edu/~hpb/Lectures/20012/LP/

http://www.owlnet.rice.edu/~comp412/Lectures/09.pdf

[8] Neff, N., *OO Design in Compiling an OO Language*, SIGCSE Bulletin, 31, 1, March 1999, 326-330

[9] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.

[10] Nguyen, D. and Wong, S. *Design Patterns for Decoupling Data Structures and Algorithms*, SIGCSE Bulletin, 31, 1, March 1999, 87-91.