# Design Patterns for Parsing

Dung "Zung" Nguyen
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-3835

dxnguyen@rice.edu

Mathias Ricken
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-3836

mgricken@rice.edu

Stephen Wong
Dept. of Computer Science
Rice University
Houston, TX 77005
+1 713-348-3814

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 present a variant form of the visitor pattern and apply it 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. For non-terminals that have more than one production rule, the corresponding visitors are chained together according to the chain of responsibility pattern in order to be processed correctly by a valid token. 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 has minimal conditionals. It not only serves to teach standard techniques in parsing but also as a non-trivial exercise of object modeling for objects-first introductory courses.

## Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming.

## General Terms

Algorithms, design, languages.

## Keywords

Parsing, grammar, CS1/CS2, objects-first, design patterns, pedagogy, modeling.

## 1   INTRODUCTION

The 2001 ACM Computing Curricula lists the objects-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 objects-first course. At this level, it is pedagogically prudent to start with a simple grammar and gradually expand it to enforce and enhance the student's understanding. Thus, it is crucial that an open-ended number of tokens, non-terminal symbols and production rules can be added to a given grammar with minimal perturbation to the existing code. 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. Such capability is effectuated via a variant of the visitor pattern [9] whose formulation will be described in Section 2.

We also need to model the non-terminal symbols of a grammar and their corresponding production rules, which define the 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.

In our parsing framework, the parsing of each non-terminal is modeled as a visitor to a token. To decouple the parsing visitors

```java
public abstract class AToken {
   private String _lexeme;
   public abstract Object execute(ITokVisitor algo, Object param);
}
// ----------------------------------------------
public interface ITokVisitor {
   public Object defaultCase(AToken host, Object param);
}
// ----------------------------------------------
public class NumToken extends AToken {

   public static interface INumVisitor extends ITokVisitor {
      public Object numCase(NumToken host, Object param);
   }

   public static abstract class AChainVis implements INumVisitor {
      private ITokVisitor _successor;
      protected AChainVis(ITokVisitor successor) {
         _successor = successor;
      }

      public Object defaultCase(AToken host, Object param) {
         return host.execute(_successor, param);
      }
   }

   public Object execute(ITokVisitor algo, Object param) {
      return (algo instanceof INumVisitor)?
          ((INumVisitor) algo).numCase(this, param):
          algo.defaultCase(this, param);
   }
}
```

**Listing 1: Modified visitor pattern for tokens (constructors omitted)**

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 4 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.

Section 5 demonstrates the flexibility and extensibility of our parsing approach with an example of how expanding the original grammar with several new tokens, non-terminals and production rules only results in minimal perturbations of the existing code.

## 2 DESIGN PATTERNS FOR TOKENS

Our object-oriented formulation of LL(1) grammars seeks to delineate and decouple the task of parsing each grammar rule for each non-terminal from the task of selecting the appropriate rule to parse. The parsing algorithm for each non-terminal X knows exactly what the rules are for X and how to proceed with each of the rules but does not know which rule to apply without identifying the current token. On the other hand, the current token intrinsically knows its type and thus can select the appropriate rule but does not know what the rule needs to do at that juncture. The visitor pattern allows these two classes of objects to cooperate to carry out the correct parsing task at the correct time without querying the current input token for its type: the tokens serve as hosts and the parsing algorithms are the visitors.

However, the standard visitor pattern requires that the number of hosts is invariant, which does not meet our design goal of being able to add an arbitrary numbers of tokens, i.e. hosts, to the system. At the core of the visitor pattern is the guarantee that any given host only calls the method of the visitor corresponding to that host. Instead of a single visitor interface that offers a fixed number of methods to a fixed number of hosts, consider a union of interfaces, one per host.

We implement this union as an interface called *ITokVisitor* with a single default case method corresponding to the abstract host, *AToken* (see Listing 1). Thus, every concrete host accepts this visitor. Each concrete host defines its own visitor sub-interface with a method specific to that host. When a concrete host accepts a visitor using *AToken*.execute(…), it checks if that visitor is its own specific visitor and calls the host-specific method of the visitor if this is the case, otherwise it calls the default case method. For instance, in Listing 1, the NumToken host checks if the visitor, algo, is of type INumVisitor. If it is, the host-specific method of the visitor, numCase(…), is called, otherwise the default case method, defaultCase(…), is called.

The visitor pattern is designed for situations where the host's type is unknown at run-time. During parsing, the current token may be one in a set of valid tokens. To enable the current token to select the appropriate visitors, we apply the chain of responsibility design pattern [9] to chain together all the visitors of the valid tokens. The assumption that the grammar is LL(1) guarantees that exactly one visitor in the chain corresponds to the current valid token. This point will be further elucidated in Section 4, where we describe the lazy manufacturing of appropriate visitors using factories.

The tokens so designed do not have any knowledge about any grammar or any action one wants to do with them. This allows the tokens from the same set to be used in different grammars. Moreover, we can add any new token to the system by sub-classing *AToken* and defining a corresponding nested sub-interface for *ITokVisitor* without affecting any of the existing code. Section 5 will further illustrated the flexibility and extensibility of our design.

We now tackle the problem of modeling the non-terminals and their production rules.

## 3 DESIGN PATTERNS FOR NON-TERMINALS AND PRODUCTION RULES

As a simple example, consider the CFG for infix arithmetic expressions using only addition, numbers and identifiers:

E :: F | F + E             F :: num | id

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

E :: F E'             F :: num | id
E' :: *empty* | + E

This grammar isn't quite ready to be modeled as classes however. This is because there is still a sequence of symbols, "+ E", that is 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 ("+ E" is replaced by E1a). Single tokens occurring in a branch also receive their own non-terminal (**num** and **id** are replaced by F1 and F2, respectively). It is clear that this is an equivalent grammar because it simply gives names to existing sequences of symbols or tokens. Below, we have changed the names slightly to remove the primes and create Java-legal symbol names:

**Figure 1**: **Object model of the example grammar.**

E :: F E1            F :: F1 | F2
E1 :: *empty* | E1a      F1 :: **num**
E1a :: **+** E          F2 :: **id**

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. We find 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 term on the right-hand side can be a representation of the left-hand side. F in the grammar above is such a branch: F1 "is-a" valid representation of F, and so is F2. On the other hand, some rules represent a sequence of terms, such as the rules E and E1a 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. E, for example, "has-a" E1 and an F.

We can now simply and directly create our object model of the grammar. In Figure 1 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 "+", identifiers and numbers, are represented by their own classes. The empty term and the end-of-file token are represented individual classes as well. 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  FACTORIES

The goal here is to maximally decouple the elements of the grammar, which will lead to a parsing system that is robust and extensible. The problem with directly defining and instantiating



**Figure 2**: **Class diagram of the factories of the token visitors.**

the parsing visitors is that at any given stage, one must analyze the details of the grammar to the level of knowing all 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 down to the F term. However, the code to process an E term should only be concerned with the fact that an E term is composed of a F term and an E1 term, and should not be coupled to the internal nature of the F, E1 or any other terms. Likewise, when parsing a branch, such as F, one must create the union of all the visitors that parse the branch's subclasses. Luckily, an LL(1) grammar insures that there are no conflicts between methods of the visitors to the branch's tokens because each token uniquely determines a sequence. However, if a branch consists of further branches, again, this would entail delving into the details of the grammar at deeper levels. In addition, the presence of loops in the grammar further complicates the analysis as it impacts the construction order of the visitors.

To remedy this problem, one must re-think the instantiation process of the visitors. In particular, in order to decouple the construction of the visitor for one term from the details of other terms, one must abstract and encapsulate the construction process. This is done by the abstract factory design pattern. Using factories to instantiate the parsing visitors

- enables each term to be decoupled from any other term by hiding the instantiation details.
- enables the construction of the union of visitors by chaining, which is used to implement branches.
- enables the lazy 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 Listing 2). All factories adhere to a basic factory interface which provides the methods to instantiate the parsing visitors. For convenience, all the factories are derived from an abstract factory, *ATVFactory*, which provides access to the tokenizer.

The factories for sequence terms (e.g. E and E1a) are initialized with the factories of their composed terms. The actual creation of the visitors is delayed until the first call to makeVisitor() or

```
// E1aFact is an inner class of EFact

// parse "+" followed by an E
private class E1aFact extends ATVFactory {
  private EFact _eFact;
  private ITokVisitor _parseE;

  private IInit _initializer = new IInit() {
    public void init() {
      _initializer = NoOpInit.Singleton; // do it only once
      _parseE = _eFact.makeVisitor(); // make visitor
    }
  };

  public E1aFact(ITokenizer tkz) { super(tkz); _eFact = EFact.this; }

  public ITokVisitor makeVisitor() {
    _initializer.init(); // lazy initialization
    return new PlusToken.ADefaultVis() {
      public Object plusCase(PlusToken host, Object inp) {
        return new E1a(host, (E) nextToken().execute(_parseE, inp));
      }
      public Object defaultCase(AToken host, Object param) {
        throw new IllegalArgumentException("Wrong token");
      }
    };
  }

  public ITokVisitor makeChainedVisitor(final ITokVisitor succ) {
    _initializer.init(); // lazy initialization
    return new PlusToken.AChainVis(succ) {
      public Object plusCase(PlusToken host, Object inp) {
        return new E1a(host, (E) nextToken().execute(_parseE, inp));
      }
    };
  }
}
```

**Listing 2: Typical factory for a sequence parsing visitor.**

makeChainedVisitor(), since only then is it guaranteed that all factories have been created and circular references can safely be established. The initializer object _initializer, which performs this lazy construction, is instantiated anonymously and replaces itself with a no-operation to ascertain it is executed only once. This is an example of the state design pattern. Listing 2 shows how the use of anonymous inner classes in the makeVisitor() method to instantiate the parsing visitor creates a closure that includes the stored visitor, _parseE.

The factories for branch terms (e.g. F, see Listing 3) are initialized with the factories for all their right-hand side terms. 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 using the chain of responsibility 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 process of creating this chain to one of the factories of the subclasses. Hence, all factories provide a method, makeChainedVisitor(*ITVFactory* succ), to produce a visitor that, in case itself is not the intended receiver (i.e. defaultCase() is called), delegates to another visitor, the successor succ, thereby establishing the chain of responsibility. The E and **empty** terms are special cases since they have defined behaviors for all token cases. Thus the visitors for these terms can only be the end of a chain.

```
// FFact is an inner class of EFact

// parse either an F1 or an F2
private class FFact extends ATVFactory {
  private F1Fact _f1Fact;
  private F2Fact _f2Fact;

  public FFact(ITokenizer tkz) {
    super(tkz);
    _f1Fact = new F1Fact(tkz);
    _f2Fact = new F2Fact(tkz);
  }

  public ITokVisitor makeVisitor() {
    return _f1Fact.makeChainedVisitor(_f2Fact.makeVisitor());
  }

  public ITokVisitor makeChainedVisitor(ITokVisitor succ) {
    return _f1Fact.makeChainedVisitor(
        _f2Fact.makeChainedVisitor(succ));
  }

  private class F1Fact extends ATVFactory { ... }
  private class F2Fact extends ATVFactory { ... }
}
```

**Listing 3: Typical factory for a branch parsing visitor.**

The result is that instead of constructing the parsing visitors directly, one now constructs the parsing visitor factories displayed in Figure 2. Note that the object structure of the factories matches that of the grammar object model in Figure 1, except that all the factory-factory relationships are compositional. 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. To start parsing, we simply ask the top-level factory, EFact, to make its visitor and apply that visitor to the first token. Note that we can avoid using mutating setXXX() methods usually necessary to produce a circular relationship if we create factories as inner classes inside the closures of other factories that use them. In Listing 2, EFact.this can be accessed directly because EFact closes over E1aFact. In Listing 3, instances of F1Fact and F2Fact can be created directly since they are implemented as inner classes of FFact. We have demonstrated this technique in an automated parser generator; unfortunately, this nesting of factory classes tends to generate a single unwieldy source file for a large grammar.

## 5 EXTENDING THE GRAMMAR

Consider a grammar that adds parenthesized expressions and multiplication to the grammar used above:

| | |
|---|---|
| E :: S E1 | T :: **num** T1 |
| E1 :: **empty** \| E1a | V :: **id** T1 |
| E1a :: **+** E | T1 :: **empty** \| T1a |
| S :: P \| T \| V | T1a :: **\*** S |
| P :: **(** E **)** | |

For this new grammar, we have changed the composition of E, added five more non-terminals, three tokens, and seven production rules, and removed a non-terminal and a rule. To parse this grammar, we have to change only two fields and one constructor parameter in the E class, and replace the F factory, FFact, with the new S factory, SFact, in the factory for E. The remaining classes can be added without modifying any of the
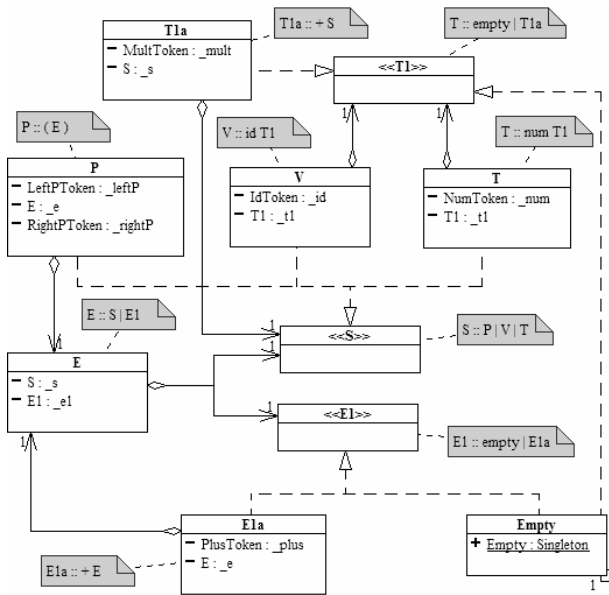
**Figure 3**: **Object model for the extended grammar.**

existing code. The UML class diagram for the terminals and non-terminals is depicted in Figure 3, the diagram for the factories can be found in Figure 4. Again, the structure of the factories matches that of the grammar.

In a procedural style it is hard to achieve the same extensibility that our object-oriented formulation exhibits. Traditionally, the addition of the "(" token as legal beginning of an expression would require a pervasive change to the existing parser. Our modified visitor pattern combined with the factory pattern provides the desired decoupling that pinpoints where the changes should be made and prevents a propagation of changes through the system.

# 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, a variant of the visitor design pattern was used to provide direct dispatching to the appropriate parsing code, thus eliminating conditionals, and to allow the open-ended addition of tokens to the grammar with minimal perturbation of the existing code. 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. The chain of responsibility pattern was 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 is an OO data structures and
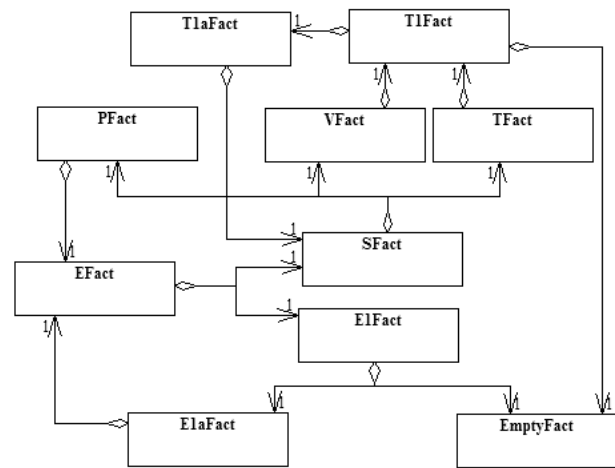


**Figure 4: Factories for the extended grammar.**

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 as a vehicle for teaching them how to decompose a problem into a 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.