ABSTRACT
A grammar is suitable for top-down recursive descent parsing if it is LL(1) [1]. Deriving the algorithm to test if a grammar is LL(1) usually involves inspection of a BNF version of the grammar, representing certain adjacency relations as matrices, and finally performing some subtle computations on the matrices. It is difficult to present the problem intuitively enough so that the algorithm can be discovered by students.

ooops [2,3] is an LL(1) parser generator that represents an EBNF-based grammar as a tree of objects that are observed as they recognize input. The objects are also able to test if the grammar is LL(1). Distributing the algorithm over several classes makes it simple enough to be discovered during a classroom presentation.

This paper discusses the new architecture of oops and the LL(1) checking and parsing algorithms and defines some extensions to EBNF which simplify some sticky language definition problems but are very simple to implement using inheritance.

2. PRINCIPLES
2.1 Grammars
A grammar consists of a start symbol, other nonterminals, terminals (input symbols), and rules. The rules of a grammar are often specified in a meta language such as BNF, e.g.:

```
bits: /*empty*/ | bits '0' | bits '1'
```

By convention, the first (or only) rule explains the start symbol. The rules of BNF itself can be expressed in BNF, for example:

```
grammar: grammar rule | rule
rule: Id ':' | alt
alt: /*empty*/ | seq | alt '|' | seq
seq: seq term | term
term: Id | String
```

Extensions to BNF usually replace term with notations for iteration in order to avoid recursive definitions as much as possible. The following form of extended BNF is often used in the Requests for Comments describing Internet protocols:

```
term: item | item '*' | item '+'
item: Id | String | '(' alt ')'
```

Suffixes limit how often a term can appear: ? permits zero or one occurrence, + requires at least one occurrence, and * allows any number of repetitions.1

2.2 Trees
A grammar produces trees. The start symbol of the grammar is used as the root of a tree. An internal node always corresponds to a nonterminal. The sequence of its direct descendants must correspond to one sequence of terms among the alternatives in the rule

---

1 To simplify recognition, processors like oops require that rules are terminated, e.g., with a semicolon. EBNF usually does not allow empty alternatives.
for that nonterminal. A leaf node always corresponds to a terminal. For example, consider a grammar for very simple arithmetic expressions:

```
sum: sum + '1' | '1'
```

Two possible trees are shown in Figure 1.

![Figure 1](image1.png)

If objects are used to represent these trees, certain leaf nodes are often omitted because their information can be represented by the class of the parent node instead. For example, simplified versions of the trees above could be constructed as

```
new Integer(1)
nw Sum(new Integer(1), new Integer(1))
```

In the case of arithmetic expressions, all node objects could be required to implement a common interface such as

```
interface Node { int intValue(); }
```

and all implementations together could act as an evaluator for arithmetic expressions.

### 2.3 Grammars as Trees

The grammars for BNF and EBNF are just grammars, i.e., they, too, will produce trees. It is therefore possible to construct a tree for a grammar which is expressed using BNF or EBNF, e.g.:

```
sum: sum + '1' | '1'
```

![Figure 2](image2.png)

Alternatively, using appropriate classes and factory methods:

```
new Grammar(
    new Rule("sum",
        new Alt(
            new Seq(new NT("sum"), new T("+"), new T("1")),
            new T("1")))))
```

For BNF the classes Grammar, Rule, Alt, Seq, NT, and T are sufficient to represent any set of rules. EBNF requires just one more class, namely Repeat; instances of Repeat hold lower and upper bounds on the number of iterations. For example

```
new Grammar(
    new Rule("sum",
        new Seq(
            new T("1"),
            new Repeat(0, INFINITY,
                new Seq("+", new T("1"))))))
```

represents

```
sum: '1' ( '+' '1' )*
```

Usually, the bounds on Repeat are only chosen from 0, 1, and infinity. However, the class is general enough to represent suffix constructs like 3..5, 7..1, or 4.. to indicate explicit iteration ranges.

### 2.4 Syntax Graphs

When describing Pascal [4] Niklaus Wirth pioneered the use of flowcharting to visualize a grammar. EBNF implies a restricted style of flowcharting which corresponds closely to the structured flowcharts proposed by Nassi and Shneiderman [5]. The structured flowcharts in Figure 3 visualize the BNF and EBNF grammars for very simple arithmetic expressions:

![Figure 3](image3.png)

The building blocks in Figure 4 represent the Grammar classes introduced in section 2.3:

```
sum       NT      Seq
\_\_\_\_   \_\_\_\_   \_\_\_\_\_\_
```

![Figure 4](image4.png)

### 3. DISCOVERIES

#### 3.1 Recognition

If an arithmetic expression is represented as a tree, it is natural to implement evaluation within the node classes. Similarly, it seems natural to require the grammar tree classes Grammar, etc. to implement language recognition.

The first algorithm P is used to parse an input file for validation against the grammar.

**Algorithm P: parsing**

- A Grammar object asks its first Rule object (the start symbol) to parse, and it expects to be at end of input when this object is done.
- A Rule object asks its descendant object to parse.
- A Seq object asks each descendant object in turn to parse.
- An Alt object selects one of its descendant objects and asks it to parse.
- A Repeat object may ask its descendant object to parse. It checks that the number of iterations is within bounds.
- A T (terminal) object checks if the next input is expected, i.e., a literal string must match exactly and a category such as
identifiers requires a suitable character sequence in the input. As a side effect of parsing a T object the input is advanced.

- A N object (nonterminal) object references another Rule by name and asks it to parse.

This algorithm can easily be “discovered” in a classroom setting. However, there seem to be two sticky spots: recognition will go astray if an Alt object does not pick the proper descendant or if a Repeat object does not start or quit iterating at the right time. These are of course the classic problems of a simple recursive descent parser, re-expressed in the context of objects.

As a remedy, a direct descendant can only be asked to parse if it (or its descendants) can consume at least the next input symbol. This means that each node has to know with which input symbols it is expected to start parsing — the lookahead of the node. This is no guarantee for success; a node may get stuck after the next symbol, but it avoids infinite loops without consuming input.

3.2 Lookahead Set
Computing the lookahead set is another task for the Grammar classes. The implementation can be “discovered”, especially if lookahead is sketched into the syntax graphs.

Algorithm L: computing the lookahead sets

- A T object trivially knows its lookahead. In fact, during parsing, a T object does not have to check the next input because it can only be asked to parse if the next input matches the lookahead, i.e., is just what the T object expects.

- A N object takes its lookahead from the Rule object in which the nonterminal appears as the left hand side.

- A Rule object must receive its lookahead from its descendant. A Grammar object must receive it from its first Rule object.

- An Alt object has as a lookahead the combination of the lookaheads of all descendants.

- A Repeat object receives its lookahead from its descendant, but there is a catch: if the lower bound on iterations is zero, the Repeat object may not have to be asked to parse at all. This can be represented by adding a special noInput symbol to the lookahead.

- A Seq object, finally, takes its lookahead from its first descendant. However, as long as noInput is a possibility, the lookaheads of further descendants must be added.

Again, there is a sticky spot: while the lookahead for the descendant of a specific Rule object is computed, a N object might be encountered that references exactly that Rule object:

\[
\text{sum: sum ' + ' '1' | '1'}
\]

but then, a left-recursive grammar is not LL(1).

Therefore, Algorithm L is used twice. During the first execution of the algorithm, each descendant of the Grammar object is marked active the first time it is visited. A Rule object must not be reached from a N object while it is active, and a Seq object proceeds only as far as necessary to determine the lookahead. This way, even subtle left recursions are discovered.

The second time, Algorithm L is applied only to the first descendant of the Grammar object, each Seq object now delegates to all descendants, and each Rule object is marked as it is encountered. This time the marks are used to find unused rules.

3.3 Checking LL(1)
The question arises if computing lookahead is sufficient so that Algorithm P will operate in a deterministic fashion. What needs to be checked is if there are objects in the tree that have a choice, for certain inputs, in which object in the tree to visit next. This can be “discovered” for each class in turn.

Algorithm C: checking LL(1)

- Grammar, Rule, T and N objects have no choice.
- An Alt object has a choice if the next input is in the lookahead of more than one descendant.
- A Repeat object has a choice if zero repetitions are allowed and the next input is in the lookahead of both the descendant of the Repeat object and whatever object follows the Repeat object.
- A Seq object has a choice if a descendant accepts noInput and the next input is in the lookahead of such a descendant and can follow the descendant, too.

The possibility of a choice is undesirable: if Algorithm P gets stuck, i.e., if the next input symbol is unacceptable, it will have to be backed up to exhaust all possibilities before it is clear that recognition is impossible.

Algorithm C can certify a grammar as deterministic in Algorithm P if there are no choices. This is easy to implement as long as the follow set of each node in a Grammar tree is known. The follow set of a node is the sum of the lookaheads of all nodes which may be asked to parse once the node is done. Therefore one more algorithm is needed.

3.4 Follow Set
Computing the follow set is the last task for the Grammar classes and can again be “discovered” for each class in turn.

Algorithm F: computing the follow sets

- Grammar and T objects do not need to know their follow set.
- An Alt object hands its follow set to each descendant.
- A Repeat object does the same but if the upper bound is more than 1 the lookahead has to be added.
- A Seq object hands the follow set to the last descendant, the lookahead set of the last descendant to the previous descendant, and so on. If a descendant accepts noInput, its follow set must be added to the follow set of the previous descendant.
- A N object sends its follow set to the corresponding Rule object.
- A Rule object sends its follow set to its descendant.

The next-to-the-last step points out a problem: the start symbol, i.e., the first descendant Rule object of the Grammar object, has
endInput as a follow set; nothing is known about the other Rule objects.

However, there is only a finite number of input symbols; therefore, to start Algorithm F the grammar object sends endInput to the first Rule object and sending continues until the follow sets of all Rules have stopped increasing.

4. OBSERVERS

If the Grammar classes are implemented as discussed above, a first recognizer for grammars can be built manually by constructing a tree for a grammar that expresses EBNF in EBNF. Algorithms L, F, and C show that the grammar is suitable so that, with a simple scanner providing input, Algorithm P can recognize the original grammar and others.

Clearly, the system is more useful if it can be used to represent recognized input as a tree, or perform some other actions during recognition. The Simple API for XML Parsing (SAX) [6] supported by Java suggests what should be done: Algorithm P needs to report to an observer.

Unlike SAX, however, oops allows for a new observer to be created whenever a Rule is asked to parse. An observer can implement the following interface:

```java
Observer init (String name, int rule);
void shift (T sender);
void shift (Observer sender);
void reduce ()
Object value ()
```

Parsing is started with an initial observer. If a Rule is selected, the current observer receives the rule name and number with init and must reply with the next observer. When a T object recognizes input the observer is informed with shift(T). The observer at that point could obtain more information from the scanner.

When a Rule is completed the current observer receives reduce and the previous observer receives shift(Observer) and is restored as the current observer.

init() can be used to send information from one observer to a new one, value() is available to retrieve information from an observer that has received reduce().

There is an ObserverAdapter available so that the actual observers only implement the methods needed to deal with the rulespecif information. The implementation of el [5] is even based on a single, general observer class which uses reflection and factory methods to construct a tree for a program.

The elegance of the observer pattern for parsing lies in the fact that for oops the grammar is purely EBNF. It does not contain any implementation language artifacts for processing recognized information. In this oops differs strongly from tools such as JavaCC[7] and ANTLR [8] where it is difficult to see and modify the grammar behind all the semantic actions.

5. EXTENSIONS

The first tree representing a grammar for EBNF has to be constructed by hand. Following that, this tree (and a matching observer) can be used to recognize the same grammar and construct a copy of itself over the Grammar classes described above.

More interesting is the fact that different versions of EBNF can be supported. One extension with numeric ranges for iterations has already been described in section 2.3.

Some authors use brackets to represent optional items and braces to group items which may be repeated. If the lower bound for a brace group is one, it makes sense to nest braces and brackets to group items that may be omitted or repeated an arbitrary number of times. This version of EBNF is expressed in the “RFC” version introduced in section Error! Reference source not found., recognized by the “first recognizer”, and a matching observer represents it over the Grammar classes. The result is a parser generator which uses a different version of EBNF for its grammars.

By convention, alternatives in a grammar are exclusive — an Alt object has to make a choice which descendant it will ask to parse. In spite of that it is customary to use the inclusive or symbol | to represent alternatives. It turns out that the familiar logical connectives of programming languages, complete with their typical precedence, can be given a useful definition in yet another extension of EBNF supported by oops:

* denotes exclusive or and is used where BNF would use |.

| denotes inclusive or and allows either or both descendants to appear — in any order.

& denotes and and requires that both descendants must appear — in any order.

The new operators allow for syntactic constraints which other systems have to implement with semantic actions. There is a slight interaction with iterations: the new operators monitor iteration bounds for any iterations that are their direct descendants so that, for example,

perm: a & b & c & ... *

describes any permutation of a, b, and c, where the elements may be separated by spaces.

The object-oriented architecture of oops is invaluable when the input language is extended because new operations can be fit into all the algorithms without affecting the implementations of the existing operations.

The class described as Alt so far is in fact xor, derived from a base class Alt which implements most of the infrastructure for alternatives; the two new operators or and and are similarly derived. The classes inherit most of the code for the checking algorithms and only need to implement their different behavior for parsing.

Another useful extension of EBNF would be a list syntax similar to the one implemented in pj [9]:

```plaintext
list: term / delim
```

This would require a subclass of Repeat so that term must appear one or more times, separated by delim.

6. SUMMARY

We have seen how a simple breakdown of parser responsibilities among a carefully designed set of classes can actually solve the common parsing challenges taught in a compiler construction course: lookahead, follow sets, grammar validation, and of course, source code recognition. The first grammar represented by the classes was the grammar of EBNF itself. From this the parser can

---

2 We are working on optionally using more lookahead. This is likely to change the interface somewhat — oops will have to cache values provided by the scanner for terminals and pass them in shift(T). The current design seems to violate the Law of Demeter.
re-generate itself and it can generate the object structure for any
grammar. Meanwhile the design of the parser is kept exposed to
the student. In studying it, he or she will also have important ob-
ject-oriented design patterns such as observer and important de-
velopment qualities such as reuse and extensibility be reinforced.
The software is freely available from the website given in the
bibliography [3].

There is a Masters thesis currently in progress wherein the student
is investigating how best to expand oops to be able to generate
LL(k) parsers.

7. REFERENCES
Verlag, 1991.
<http://www.saxproject.org/>.
<http://www.cs.rit.edu/~ats/projects/lp/doc/pj/package-
summary.html>.