Rochester Institute of Technology

RIT Scholar Works

Articles | Faculty & Staff Scholarship

2000

An Object-oriented LL(1) parser generator

Bernd Kuhl

Axel-Tobias Schreiner

Follow this and additional works at: https://scholarworks.rit.edu/article

Recommended Citation

Kuhl, Bernd and Schreiner, Axel-Tobias, "An Object-oriented LL(1) parser generator" (2000). *ACM SIGPLAN Notices*, Vol. 35 (No. 12),Accessed from
https://scholarworks.rit.edu/article/655

This Article is brought to you for free and open access by the Faculty & Staff Scholarship at RIT Scholar Works. It has been accepted for inclusion in Articles by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

Copyright ACM, 2000. This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in ACM SIGPLAN Notices, Volume 35, Issue 12 (December 2000). http://doi.acm.org/10.1145/369928.369941

An object-oriented LL(1) parser generator

Bernd Kühl and Axel-Tobias Schreiner
{Bernd.Kuehl,Axel.Schreiner}@informatik.uni-osnabrueck.de
Computer Science, University of Osnabrück, Germany

Abstract

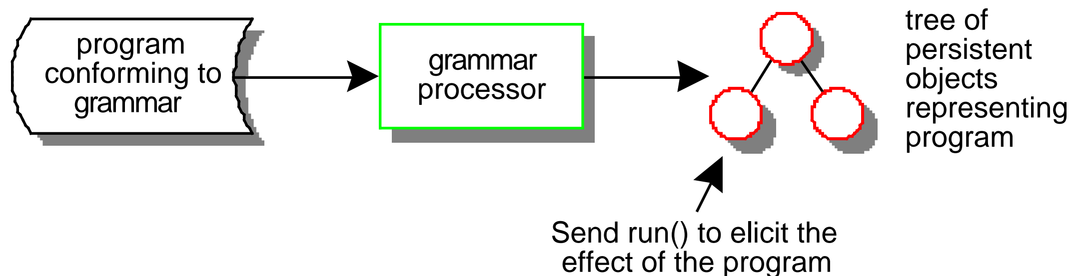
This paper describes *oops*, an object-oriented parser generator implemented in Java [1]. *Oops* takes a grammar written in EBNF, checks that it is indeed LL(1), i.e., suitable for recursive descent parsing, and produces a parser as a set of serialized objects. A scanner must be provided and classes satisfying certain interfaces can be implemented which the parser uses to build parse trees.

The paper discusses the ideas behind *oops* — which are not specific to an implementation in Java — and shows the advantages of an object-oriented approach to grammar verification and parsing.

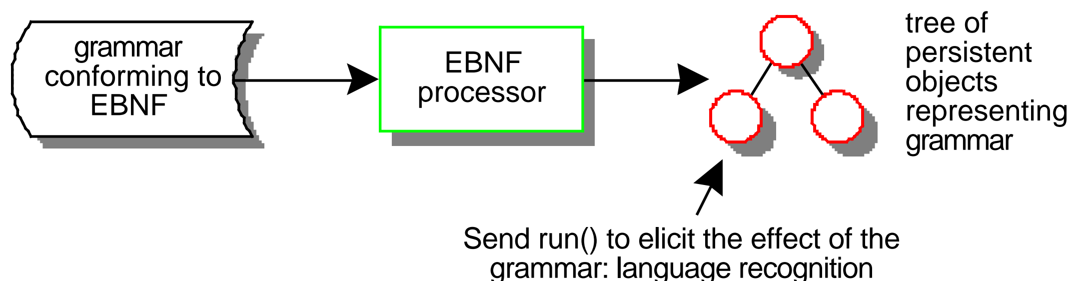
Keywords: compiler, parser, object-orientation

Objects for analysis and execution

A compiler converts a program written in a language into a tree which can be interpreted or synthesized for some target machine. Execution of the result, tree or machine code, performs the task for which the program was written.



This idea can be applied to grammars and parsers. A parser generator converts a grammar written in a meta-language into a tree. Execution of such a tree should perform what the grammar is intended for, i.e., language recognition. From this point of view the tree constitutes a parser.



There must be semantic analysis in both cases. If a program is converted into a tree, semantic analysis checks the validity of the program. If a grammar is converted into a tree, semantic analysis must be concerned with the suitability of the grammar for language recognition.

An object-oriented approach builds both trees from objects. Semantic analysis amounts to sending a message to the root of a tree asking it to check it's validity. The root will in turn activate it's subtrees and so on, i.e., semantic analysis is now a localized problem applied to each class of tree node.

Executing a tree can also be accomplished by messaging the root of the tree. The root will enlist the help of it's subtrees and thus execution is similarly distributed over the tree node classes.

Mapping a grammar to a set of objects

A context-free grammar consists of nonterminals, terminals, a nonterminal start symbol, and rules describing how a nonterminal produces various sequences consisting of terminals and nonterminals. Using a typical variant of BNF, a rule looks like this:

```
identifier : letter | identifier letter | identifier number
```

Recursion is used to express iteration. So-called Extended BNF provides notations to express this more directly.[1]

```
identifier : letter [{ letter | number }]
```

This notation eliminates most recursions and the need for empty alternatives.

Turning now from EBNF to objects, the grammar is represented as a `Parser` object containing a list of rules.

Each rule is represented as a `Rule` object connecting the nonterminal on the left hand side to the alternative symbol sequences on the right hand side.

The right hand side of each rule is a single object, i.e., a sequence of symbols is contained in a `Seq` object. Alternatives are collected in an `Alt` object. For the iterations we use `Some` and `Opt` objects and the special case of nesting `Some` and `Opt` as above is addressed by `Many`.

A nonterminal on the right hand side is represented as an `Id` object that references the rule for the nonterminal. A terminal is represented as a `Lit` object literally representing a operator or keyword such as `<=` or `begin`. User-definable terminals such as names or various numbers are represented as `Token` objects.

Using indentation to express node containment, the example above results in the following tree:

```
Parser
   Rule identifier
      Seq
         Id letter
         Many
            Alt
               Id letter
               Id number
```

Executing a Parser

A parser takes a string of terminals and decides if it is a sentence of a grammar, i.e., if the start symbol of the grammar produces the terminal string.

One way to decide is by using recursive descent. In object oriented terms the `Parser` asks the `Rule` for the start symbol and it asks it's right-hand side. An `Alt` object asks the alternatives; a `Seq` asks each constituent in turn; `Some`, `Opt`, and `Many` manage repeated queries; an `Id` invokes it's `Rule`; and a `Token` or a `Lit`, finally, checks the input. Each class, therefore, implements a `parse()`-Method and the methods call each other recursive-descent-style.

---

1. The representations vary. We prefer braces { } for something that occurs once or more often and brackets `[ ]` for something that is optional. Nesting the two, something can occur zero or more times.

It helps if this is done in a very deterministic fashion: `Alt` should only ask the appropriate alternative; `Some` and the others should only iterate if necessary.

The key to determinism are lookahead sets. In object-oriented terms, each object knows it's lookahead set of terminal symbols. The lookahead set contains all input symbols of which one must be present for the object to succeed in recognizing a sentence. We will show below, how lookahead sets are computed.

Given the lookahead sets, each object asked to participate in recognition can easily decide if it has a chance to succeed: the current input symbol must be in it's lookahead. `Rule` only starts, if the input fits. `Alt` can deterministically select a single alternative if the lookaheads of it's constituents differ. `Seq` simply asks one object after another to participate. `Id` defers to it's `Rule` and `Token` or `Lit`, after all, must match the input and advance to the next symbol.

Iterations are curtailed by lookahead, too; e.g., `Some` requires the current input to match the lookahead of it's constitutent at least once. All iterators stop iterating, once the input does not match anymore, or not at all.

Suitable grammars

A parser decides if an input string is a sentence of a grammar by deriving the string from the start symbol of the grammar, i.e., by building a parse tree with the start symbol as the root and the input symbols as the leaves. Usually meaning is associated with this tree; therefore, compiler makers require a grammar to be unambiguous, i.e., to produce but a single tree for each sentence.

In general it is difficult to decide if a grammar is unambiguous. However, we saw above that the lookahead sets are the key to determinism and therefore absence of ambiguity. The approach can backfire, however. Consider

```
bits : { 0 | 1 } ( 0 | 1 )
```

where the parentheses are used for precedence. `bits` are strings of two or more zeroes or ones. The right hand side of the `Rule` for `bits` is as follows, with the interesting lookaheads listed in the middle:

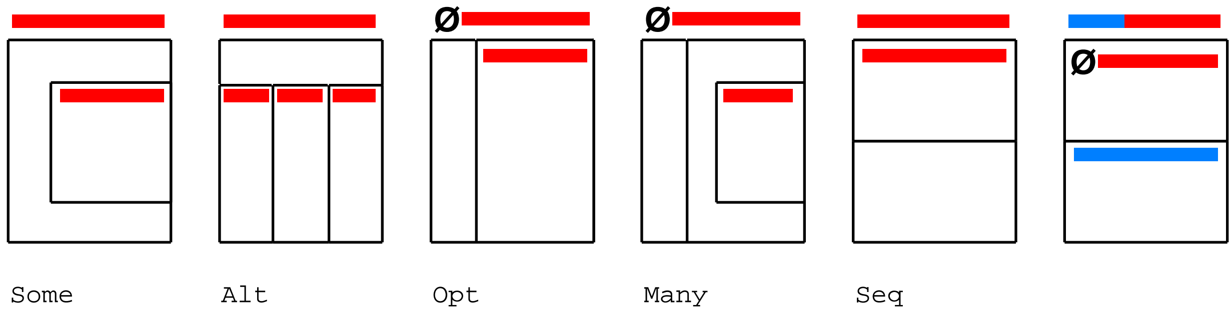
```
seq                0 1          [empty]
  some             0 1          0 1
    alt            0 1          0 1
      lit 0        0            0 1
      lit 1        1            0 1
  alt              0 1          [empty]
    lit 0          0            [empty]
    lit 1          1            [empty]
```

`Some` clearly will absorb as many `0` and `1` inputs as there are, with none left for the subsequent (not constituent!) `Alt`.

While this grammar is unambiguous, it is clearly not suitable for the parsing technique described above. For a valid parser, we need to check the follow sets, too. In object-oriented terms, the follow set contains all those input symbols one of which must be present after the input sequence recognized by the object. The follow sets are listed at right above.

Lookahead sets

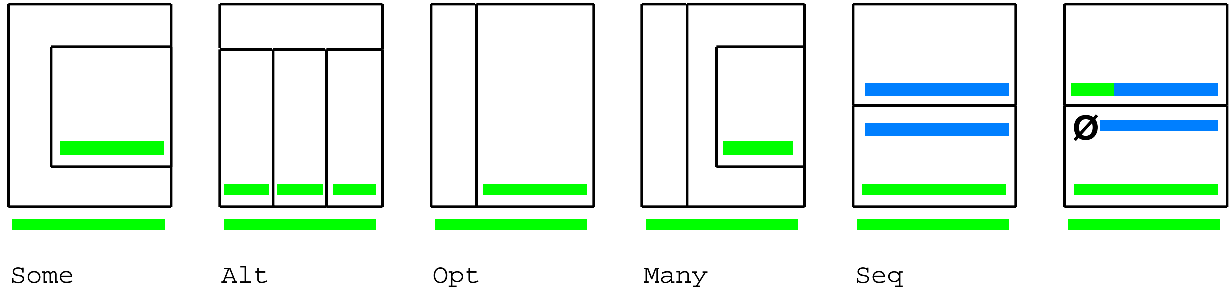
The rules for computing lookahead and follow sets can be derived mostly by considering syntax graphs as introduced by Wirth. Using a notation inspired by Nassi-Shneiderman's flowcharts[2] the interesting tree nodes look as follows:



Clearly, `Some` takes it's lookahead set from it's constituent. `Alt`'s lookahead is the union of the lookaheads of it's alternatives. `Opt` need not accept any input; therefore, it must add an empty input to the lookahead of it's constituent. `Many` is the combination of `Some` and `Opt`. Finally, `Seq` takes the lookahead of it's first entry and adds the lookaheads of subsequent entries, as long as the empty input is acceptable.

`Lit` and `Token`'s lookahead sets are trivial, they simply contain the expected input symbol. These sets kick off the rest of the computation.

Follow sets



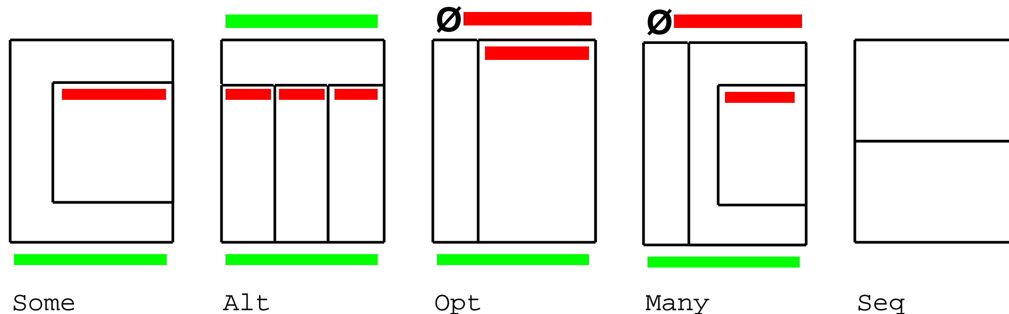
`Some`, `Alt`, `Opt`, and `Many` pass their follow set to their constituents. `Seq` passes it's follow set to the last entry. From then on, the lookahead set of an entry is the follow set of the previous entry; however, if a lookahead includes the empty input, the follow set is also passed to the previous entry.

On the whole, computing lookaheads is easy because they eventually stem from the terminals, i.e., from `Lit` and `Token`, a `Rule` gets it's lookahead from the right hand side, and an `Id` refers to it's `Rule`.

When computing follow sets, the `Rule` for the grammar's start symbol initially passes a follow set just containing an end of file to it's right hand side to start the ball rolling. When an `Id` receives a follow set, it hands it to it's `Rule`. The process is repeated as long as the follow set of any `Rule` increases; it must terminate because there is only a finite number of terminal symbols. Note that a follow set cannot contain an empty input but one or more follow sets will contain end of file. While computing follow sets is not quite as straightforward as computing lookaheads, it is a consequence of needing the transitive closure of the immediate follow relation defined by the right hand side of rules which is usually computed with Warshall's algorithm.

Checking a grammar

Obviously, a `Some` is in trouble if it's lookahead set and it's follow set have a non-empty intersection — just consider the syntax graphs:



Once Some's constituent has been satisfied, the parser checks the next input symbol to decide if it should try the constitutent again or finish with Some. This can only be decided if the two sets have no elements in common.

For an Alt, the alternatives' lookaheads must be pairwise disjoint. Thus, at most one alternative can admit the empty input; if it does, Alt's total lookahead must be disjoint from it's follow set. Because Opt and Many accept the emtpy input, their lookaheads cannot have elements in common with their follow sets, either.

The remaining building blocks, Seq, Id, Token, and Lit, impose no restrictions.

In order to be suitable for the parsing technique described above, a grammar has to satisfy the following three properties: starting with the right hand side of the rule for the start symbol, all rules must be reachable; the constituents must satisfy the restrictions outlined above; and there must not be an infinite recursion. A grammar is checked by asking it's start Rule to check it's right hand side and all classes implement the necessary check method. Infinite recursion and reachability of all rules are detected by marking algorithms.

The elegance of the object-oriented approach is threefold: Computing lookahead and follow sets is a local, simple algorithm for each class. The lookahead sets can be carried over to execution as described previously. The follow sets can be useful for an automated error recovery technique that will be described in another paper.

Using oops

Oops takes a grammar specified in EBNF and represents it as a collection of Rule objects containing right hand sides modeled using the other classes introduced above. The objects are then asked to check if the grammar is suitable for recognition; as a side effect they will then contain the lookahead and follow sets. The result is serialized.

Once the objects are revived, they can be used to parse a sentence over the grammar: The starting Rule is given a scanner representing input symbols as singletons and it asks it's right hand side to parse. Because each object contains it's lookahead set, it is quite simple to check if the input matches. At the end, the starting Rule checks with the scanner if the end of the input has indeed been reached.

While this may be very satisfying from a theoretical standpoint, the user usually expects the parser to return a bit more than a simple true or false. Therefore, *oops* provides interfaces for building parse trees or executing other actions.

While a Rule matches something, a subtree of the eventual parse tree can be built: The Rule creates a Goal object and passes it to the right hand side when the parse is started and the Goal is handed off to each parser object. If a Lit or Token object is successful, it informs the Goal object using a shift() method which may choose to add information to a tree. Once the end of a Rule is reached, the Rule itself sends reduce() to the Goal signalling completion. The result returned by reduce() is sent with shift() to the Goal object for which the Rule was activated.

Goal is an interface and we provide a GoalAdapter and a GoalDebugger as default implementations. Both simply return the object which the first Token or Rule passes to them; GoalDebugger additionally

writes a trace. While this alone does not produce a parse tree, the trace does, however, suffice to debug a grammar.

`GoalAdapter` is supposed to be extended. When compiling the parser, *oops* checks for each `Rule` if it can find a class based on the name of the `Rule`'s nonterminal. An instance of this class is created whenever the `Rule` needs another `Goal` object. If a `Rule`-specific class cannot be found, `GoalAdapter` or some other default class is used. A trivial tree builder extends `GoalAdapter` and overwrites `reduce()` and `shift()` in some critical places to construct a more useful tree. Continuing with the previous example:

```
identifier : letter [{ letter | number }]
```

If the parser really is to assemble an `identifier`, a class `identifier` must be implemented:

```
public class identifier extends GoalAdapter {
    StringBuffer result = new StringBuffer();
    public void shift (Token sender, Object node) {
        if (node != null)
          result.append(node);  // a Character from the scanner
    }
    public Object reduce () {
        return result.toString();
    }
}
```

`letter` and `number` must be recognized as `Token` objects by a scanner. `Token` denotes a category of inputs; therefore, the scanner would pass a `Character` object to describe the actual input symbol. `identifier` receives the `Token` and can store the accompanying object as recognition proceeds along the right hand side of the rule — this is equivalent to *yacc*'s value stack. Once the rule is completed, `identifier` is asked to return an object corresponding to the nonterminal, to which the right hand side is now reduced. The object is handed off to the superior `Goal` using another `shift()` method. Consider:

```
identifiers: { identifier "\n" } ;
```

The class `identifiers` could be implemented as follows:

```
public class identifiers extends GoalAdapter {
    public void shift (Goal sender, Object node) {
        if (node != null)
            System.out.println("\tid: "+node);
    }
    public void shift (Lit sender) {
        // ignore \n
    }
}
```

In this case `reduce()` returns `null` by default — the parser displays the identifier names and ignores the line separators which the scanner delivers as `Lit` objects to the parser.

`Goal` can actually be simplified. If a `Rule`-specific class implements the `Reduce` interface, the `Rule` creates a `GoalReducer` object which collects all information arriving with any `shift()` method. Once `reduce()` is finally sent to the `GoalReducer` it passes an array with the collected information to the `reduce()` method in the `Reduce` interface implemented by the `Rule`-specific class. `Reduce` is helpful for quick and dirty implementations of abstract parse trees.

Implementation issues

Oops is implemented in Java; however, the ideas do not rely on Java for the most part. Grammar and tree building actions are completely separate. The grammar is specified using some variant of EBNF; it remains unchanged no matter which language *oops* is implemented in. Actions are implemented as `shift()` and

`reduce()` methods of `Goal` or `Reduce` classes. This in turn is completely free of artifacts associated with the grammar representation.

The implementation language should provide for serialization. If it does not, *oops* will have to reread the grammar, first to send a verification message, and later to send parse messages. At least the lookaheads must be computed in both cases.

If the implementation language does not provide for interfaces, the action classes end up with a common ancestor and the scanner, too, must be derived from a class prescribed by *oops*.

Needless to say, *oops* is implemented in *oops*. We used *jay*, our Java-targeted version of *yacc* [3, 4], for the initial bootstrap.

Conclusion

Oops was patterned after Wirth's *generalparser*, a grammar graph traversal program designed to illustrate how language recognition is accomplished [5]. If implemented in an object-oriented language, the graph nodes are objects and traversal can be delegated to methods directly associated with the nodes. Wirth's *generalparser* accepts an arbitrary grammar — it is up to the user to supply a suitable grammar so that the traversal works out.

A recent paper in SIGPLAN Notices [6] reports on systematically building a recursive descent parser by hand from a grammar specified in Greibach Normal Form [7]. While this form ensures that each rule starts with a terminal symbol and thus precludes left recursion, potential ambiguities in the grammar must be dealt with each time when the parser is executed.

Oops, like *generalparser*, generates a parser mechanically. *Oops*, however, rejects an unsuitable grammar because it is a natural question to ask our graph nodes to verify the suitability of their arrangement.

Oops naturally leads to the discovery of the parsing and checking algorithms which turn out to be localized to the classes making up the graph nodes. Object orientation results in a divide and conquer approach to grammar verification and parsing.

Verification requires the computation of the lookahead and follow sets. Divide and conquer means that each class only needs to determine it's own and this can be done mostly by direct inspection, i.e., the algorithms can easily be “discovered” in a lecture situation. Even the transitive closure required for the follow sets can be discovered. After all, a `Rule` must import it's follow set from it's successor, and as we discover more follow situations within `Seq`, we have to iterate the computation for the `Rule` objects.

Objects encapsulate state, in our case, each node knows it's lookahead and follow set as part of the verification process. *Oops* reuses this information during parsing, and this, too, can be “discovered”. Lookahead sets control the direction of the traversal for parsing and follow sets can come in handy when there is an input error to decide how to continue parsing. The localized, object-oriented approach closely integrates verification and parsing and provides insight into both algorithms.

Oops requires a grammar to be LL(1) and as such does not solve the dangling else problem. Verification need not be so strict, however, and it is a simple, localized exercise to change verification to deal with this kind of conflict.

Parser generators mostly suffer from a rather baroque input syntax: grammar patterns have to be intermixed with programming language statements to define actions which take place if the patterns are matched. *Oops* completely separates the two: a grammar is specified using only EBNF and the actions are methods of classes matched by nonterminal names and implementing the `Goal` or `Reduce` interfaces; no special syntax is required for the actions or to match classes to nonterminals. In that respect, *oops* is implementation-language-independent. Different actions can be attached to the same grammar simply by selecting different libraries when the parser is started.

Yacc uses BNF to clearly associate a reduce operation and action with a nonterminal. BNF needs recursion to express iteration and left recursion precludes the use of recursive descent for recognition.

Recursive descent recognizers like *oops* and *JavaCC* [8] use EBNF to express a grammar because EBNF permits iteration and helps to avoid left recursion. EBNF, however, has a significant drawback: there is no easy way to extend *yacc*'s syntax `$i` to symbols shifted several times around a loop, i.e., if grammar and actions are mixed in the same source, the actions will have to be embedded inside the iterations in EBNF.

`Goal` overcomes this drawback: a `Goal` object is created when a `Rule` is entered. This object is informed by a `shift`-message whenever a symbol required by the `Rule` is matched; therefore, it has a sequential view of the matching process even through iterations.

Oops reuses many components. There is one set of building blocks for all generated parsers. `Goal-Adapter` is a generic building block for parse trees so that a grammar can be checked out immediately. Further building blocks for parse trees can be provided if grammars stick to fairly standard nonterminal names — in fact, we are connecting a ``compiler kit'' to *oops*, i.e., a library for parse trees that includes representations for data types, variables, control structures, semantic checks, and interpreter generation.

References

[1] Homepage *oops*: `http://www.informatik.uni-osnabrueck.de/bernd/oops/`

[2] Nassi, I., Shneiderman, B., Flowchart techniques for structured programming, SIGPLAN Notices, ACM August 1973.

[3] Kühl, B., Schreiner, A.-T., jay – Compiler bauen mit yacc und Java, iX 10/99, Heise Verlag, Germany.

[4] Homepage *jay*: `http://www.informatik.uni-osnabrueck.de/bernd/jay/`

[5] Wirth, N., Compilerbau, 3. Auflage, Teubner, 1984

[6] Davis, M. S., An object-oriented approach to constructing recursive descent parsers, SIGPLAN Notices, ACM February 2000.

[7] Greibach, S., A new normal form theorem for context-free phrase structure grammars, JACM January 12 (1) 1965.

[8] Homepages *JavaCC* and Metamata *Parse*: `http://www.metamata.com/`.