2002

Objects for lexical analysis

Bernd Kuhl

Axel-Tobias Schreiner

Objects for Lexical Analysis

Bernd Kühl
bernd@informatik.uni-osnabrueck.de
Computer Science, University of Osnabrück, Germany

Axel-Tobias Schreiner
ats@cs.rit.edu
Department of Computer Science, Rochester Institute of Technology, USA

Abstract

This paper presents a new idea for lexical analysis: *lolo* (language-oriented lexer objects) is strictly based on the object orientation paradigm. We introduce the idea behind the system, describe the implementation, and compare it to the conventional approach using *lex*[1] or *flex*[2].

lolo[3] extracts symbols from a sequence of input characters belonging to the ASCII or Unicode sets. *lolo* scanners can be extended without access to the source code: symbol recognizers can be derived by inheritance and an executing scanner can be reconfigured for different contexts. Recognizer actions are represented by objects which may be replaced at any time. Recognizers need not be based on finite state automata; therefore, *lolo* can recognize symbols that systems like *lex* cannot recognize directly.

The idea

Conventional tools for lexical analysis, such as *lex*, partition text based on patterns, i.e., on regular expressions. Patterns are associated with program statements to be executed upon recognition.

Pattern syntax is rather cryptic and difficult for novices to understand. To understand a complicated older pattern (or even someone else's) usually requires considerable experience. As an example, here is a regular expression for a comment based on C conventions:

```
"/*"([^*]|"*"+[^/*])*"*"+"/"
```

A complicated regular expression is error prone and requires extensive testing.

Tools such as *lex* process many pattern-action pairs and generate source code for a scanner to perform the lexical analysis described by the pattern-action pairs. The scanner can only be used after further compilation. The entire development cycle — editing pattern-action pairs, generating source code, and compilation — must be repeated to correct, change, or extend the scanner.

Some real-world recognition problems are difficult or even impossible to solve using regular expressions, e.g., nested comments.

These observations motivated a new and simpler approach:

A scanner is modeled as a competition of objects, each concerned with recognizing a single symbol. A room filled with such objects obtains input and sends one input character after another to the objects (push principle). An object leaves the room once it cannot deal with a character. The winner is the last object to leave the room; it has recognized the longest possible character sequence. If several objects together are last to leave, there is an ambiguity which can be resolved in favour of the first object to enter the room.

This is a simple approach for the end user. There is a class library with typical, configurable symbol recognizers such as different numerical literals, character sets and sequences, comments, white space, etc. For good measure one could even include a class with regular expressions for symbol recognition. The user just assembles objects in the room and does not have to worry about the recognition process carried out by the individual objects.

Objects encapsulate arbitrary state; therefore, these objects can be more powerful recognizers than finite state automata. This permits simple recognition of things like nested comments. While the individual recognizers are small, problem-specific automata, the room combines them into a larger, non-deterministic system.

lex-like tools first build a non-deterministic, finite state automaton from text patterns: each input causes a transition from a state to one or more other states. Sets of new states are then used as states of an equivalent, much larger, deterministic automaton, which essentially traces through all possibilities of the original automaton in parallel. The new automaton must be reduced and compiled, however.

lolo avoids this because the recognizers save the states locally and the competition essentially operates them in parallel. While such a system is simple to use and extend, even based on previously compiled code, we found that it incurrs a significant performance penalty. Luckily, for the typical problem of analyzing programming languages the first character of a symbol tends to be significant, e.g., a digit starts a number, a quote starts a string, a letter starts a reserved word or identifier (which are later differentiated by the symbol table), and special characters tend to be used singly or in pairs, etc. Thus, the outcome of the competition can usually be decided by looking up the winning recognizer in a table indexed by one character and this recognizer collects as many more characters as it can (pull principle). A special multiplexing recognizer can be entered in the table as a proxy if the first character should not be sufficient to decide part of the competition.

Partitioning a text does not make much sense unless there are actions to deal with the pieces. A recognizer that wins the competition usually knows another object that is then asked to deal with the symbol. Only the interface between the

recognizer and the action object is defined; the action object must be supplied by the user and it can be replaced at any time.

We will first present the implementation of the system, as a background to discussing the pros and cons of the technique.

Implementation

lolo is based on the collection classes introduced with version 1.2 of Sun's Java Development Kit. A scanner cannot be used simultaneously by parallel threads.

The package `lolo` provides a framework for the competition:

The class `Input` buffers characters from a `Reader` and manages the currently recognized character sequence. An `Input` serves a `Scanner`. A `Scanner` calls `setStartSymbol()` to let `Input` note the beginning of a new symbol in the buffer. `next()` returns a character from the buffer and moves the current position. `mark()` asks `Input` to flag a position in the buffer so that `pushBackToMarked()` can reset the current position.

A `Scanner` manages a recognizer table, is constructed from `Scan` objects, i.e., recognizers, and has methods to add or remove them. In response to `pack()` or just before the first `scan()`, a `Scanner` builds it's recognizer table by sending each index character to each recognizer and inserting them directly into the table or into `Mux` objects as required. `scan()` sends an `Input` to a `Scanner` and returns the winning `Scan` object or `null` if `Input` reaches end of file. `Input` is responsible for silently maintaining and filling a sufficiently large buffer.

`Scan` is the abstract base class for all recognizers. If `scan()` is sent to a `Scanner`, it selects the winning recognizer from it's table and pushes characters from `Input` to this `Scan` by calling `nextChar()`. The resulting `Scan.State` reflects in a boolean value `found`, if this character could complete a symbol and in a boolean value `more`, if more characters can be added to the symbol. `Scanner` uses `Scan.State` to mark the `Input` and terminate pushing characters.

A `Scan` object can be marked to be ignored. If it is not, and if it wins the competition, `Scanner` sends `action()` with an `Input` buffer segment containing the recognized character sequence. `Scan` delegates this message to an object supplied by the user which must implement `Scan.Action`. A recognizer is `reset()` before it is used again.

The package `lolo.scans` contains recognizers for typical programming language symbols: identifiers, numbers, strings, comments, whitespace, etc. The user can use these, extend them, or provide completely new ones. Additionally, `lolo.Mux` can be used to bundle several recognizers into a single one.

Examples

The user interacts with *lolo* in a few steps: first, a `Scanner` is constructed from a set of recognizers, which might include extended or even new ones; second, action objects can be created and connected to the recognizers; third, the `Scanner` table can be packed and the `Scanner` serialized to be used later. Eventually, an `Input` is constructed from a `Reader` and a `Scanner` is instructed with `scan()` to analyze it, which results in calls to the action objects.

Here is how some recognizers are implemented:

`Set` objects recognize a single character that does or does not belong to a set of characters:

```
public class Set extends Scan {
     protected final String set; protected final boolean inside;

     public Set(String set, boolean inside) {
          this.set = set; this.inside = inside;
     }

     public void reset() {}

     public State nextChar(char ch) {
          return stateObject.set(
          false,                          // more characters ?
          inside ?                        // found symbol ?
               set.indexOf(ch) >= 0 :
               set.indexOf(ch) < 0);
     }
}
```

`stateObject` is defined in `Scan` as a `convenience`. For `Set` it marks that no further characters are needed and `indexOf()` checks whether `Set` found a symbol.

Even this simple class can be extended: a `Char` object recognizes a single specific character or a single arbitrary character:

```
public class Char extends Set {
     public Char() { super("", false); }            // match any
     public Char(char ch) { super(ch+"", true); }   // match ch
}
```

`SetMN` is similar to `Set`: it recognizes between `m` and `n` characters which do or do not belong to a set:

```
public class SetMN extends Set {
     protected final int m, n;
     public SetMN(String set, boolean inside, int m, int n) {
          super(set, inside);
          this.m = m; this.n = n;
     }

     protected transient int jog;

     public void reset() { jog = 0; }

     public State nextChar(char ch) {
          jog ++;
          boolean b = inside ?
               set.indexOf(ch) >= 0 :
               set.indexOf(ch) < 0;
          return stateObject.set(
               b && jog < n,      // more characters ?
               b && jog >= m      // found symbol ?
          );
     }
}
```

Another class, `Int`, recognizes unsigned integer numbers. `Int` is derived from `SetMN`. One `Int` object recognizes at least a single digit:

```
public class Int extends SetMN {
     public Int() { this(Integer.MAX_VALUE); }
     public Int(int maxDigits) {
          super("0123456789", true, 1, maxDigits);
     }
}
```

`Word` objects recognize character sequences:

```
public class Word extends Scan {
     protected final String word;
     protected transient int index;

     public Word(String word) { this.word = word; }

     public void reset() { index = 0; }

     public State nextChar(char ch) {
          boolean b = ch == word.charAt(index++);
          return stateObject.set(
               b && index < word.length(), // more characters ?
               b && index == word.length() // found symbol ?
          );
     }
}
```

Serialization

`Scanner` and `Scan` implement the interface `Serializable`. `Scan` serializes an action object if it is `Serializable`. The classes implemented thus far ensure that only necessary instance variables, such as the character sequence but not `index` in `Word`, are serialized.

A `Scanner` can be stored as a collection of serialized objects so that it's table is already packed whenever the scanner is read in to be executed. Conceivably, different scanners might even share the same serialized objects.

Analysis

It seems that *lolo*'s object oriented approach to scanner construction offers numerous advantages and has but a few drawbacks.

If `lolo.scans` does not provide a suitable subclass of `Scan` one has to be implemented. This is likely to take longer than just adding a regular expression to a lex program. However, given a reasonable recognizer library, adding from scratch should hardly be necessary.

The user has to understand which subclasses of `Scan` are available. Learning how to use a tool like *lex* and the subtleties of regular expressions is likely to take longer.

An *lolo* scanner needs the class library at runtime, a *lex* scanner technically is a stand-alone piece of software. However, the class libraries are simply another system archive that must be added to the class path, or they could be installed as a Java extension.

Performance issues are the only drawback. A *lex* scanner is a deterministic finite state automaton described by a state table. An *lolo* scanner using `Mux` is not deterministic and follows a few possibilities in parallel. Fortunately, this scenario is quite unlikely when programming language sources are analyzed using suitable recognizers. Unlike *lex*, *lolo* needs to `reset()` recognizers before they can be started again.

JLex[4,5] implements a *lex* scanner in Java. Looking for typical symbols in a Java source file with 2.16 MB on different machines and on different operating systems took from1.46 to 2.37 times longer using *lolo* than using *JLex*. If an action was included to print the recognized symbols to standard output it toke only 1.00 - 1.19 times longer.

lolo's object oriented approach has a lot of advantages and some additional features. It is certainly more intuitive to be used with object-oriented languages such as Java, Objective C or C++.

`Scanner` and `Input` are tied very loosely. A `Scanner` can be called with a different `Input` for each symbol, and the scanner table could even be changed right before a symbol is scanned. It is cheaper, however, to simply add the same recognizers to different `Scanner` objects to model different scanning situations. This corresponds to defining states and adding new regular expressions to a *lex* table; however, for *lex* it requires a complete development cycle — edit, preprocess, compile — before the revised scanner can be used.

Action objects can be changed on the fly. A *lex* action would have to be explicitly based on an object reference to provide the same flexibility.

The class library contains simple but powerful recognizer classes. For *lex* one can only provide a cookbook of regular expressions which are necessarily cryptic and not necessarily copied correctly. The classes, however, can be debugged once and for all.

Library classes can be reused between scanners, regular expressions have to be copied manually.

A vendor can provide it's own class library for *lolo*. No sources are required to use and extend a class library. Regular expressions can only be sold as sources.

Subclassing can be used to specialize existing classes such as `Char`. Rather than declaring the first longest match the winner, a subclass of `Mux` could decide on a different approach to resolving ambiguity. This is not possible for *lex*.

`Input` uses Unicode characters, a `Scanner` table can be based on Unicode or `ASCII`. If *JLex* is forced to use Unicode it takes much longer to create the scanner; moreover, it is at least difficult to write patterns involving non-ASCII characters.

`Scan` and `Scanner` objects are serializable. This provides a very cheap way to reuse the same scanner objects for different projects and even on different platforms — no recompilation of the scanner is required.

Objects encapsulate state; therefore, `Scan` objects can be more powerful recognizers than plain finite state automata, e.g. the recognition of nested comments.

Conclusion

lolo scanners are useful for integration with parsers generated by systems like *oops*[6] or *jay*[7,8]. (*oops* is a strictly object-oriented parser generator which we implemented in Java, *jay* is our port of *yacc* for Java.)

We think that *lolo* provides many advantages over purely regular expression based scanners.There is a small performance penalty, but it is justified by flexibility, a shorter development cycle and more features. There is already a library of recognizers for most typical symbol patterns. Some of these recognizers are more powerful than regular expressions.

To summarize, *lolo* is in it's own class when it comes to lexical analysis.

References

[1] lex [Lesk 1975] M. E. Lesk, Lex - a lexical analyzer generator, Tech. Rep. Computing Science, Technical Report 39, Bell Laboratories, 1975.

[2] V. Paxson, Flex - Fast lexical analyzer generator, Lawrence Berkeley Laboratory, `ftp://ftp.ee.lbl.gov/flex-2.5.4.tar.gz`, 1995.

[3] Homepage *lolo*:
`http://www.inf.uos.de/bernd/lolo`

[4] Homepage JLex:
`http://www.cs.princeton.edu/~appel/modern/java/JLex/`

[5] Andrew W. Appel, Modern compiler implementation in Java, Cambridge University Press, 1997.

[6] Homepage oops:
`http://www.inf.uos.de/oops/`

[7] jay -- Compiler bauen mit yacc und Java, iX 10/99, Heise Verlag, Germany.

[8] Homepage jay:
`http://www.inf.uos.de/jay/`

[9] Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman, Compilers -- Principles, Techniques and Tools, Bell Laboratories, 1986/87.