Objects for Lexical Analysis

Bernd Kühl and Axel-Tobias Schreiner
{bekuehl,axel}@uos.de
Computer Science, University of Osnabrück, Germany

Abstract

This paper presents a new idea for lexical analysis: *oolex* (object-oriented lexer) 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 *flex* or *lex*.

oolex extracts symbols from a sequence of Unicode input characters. It 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. *oolex* need not be based on finite state automata; therefore, it can recognize symbols that systems like *flex* cannot recognize directly. *oolex* can be used for rapid prototyping: most of the existing recognizers can represent themselves as regular expressions for the Java based *JLex*.

The idea

Conventional tools for lexical analysis, such as *flex* or *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 *flex* process many pattern-action pairs and generate source code for a scanner based on the lexical analysis described by the pattern-action pairs. The scanner can only be used after further compilation. The entire development cycle — pattern-action pairs, source code generation, 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., C-style comments or even nested comments.

These considerations motivated a new and simpler approach:

A scanner is modeled as a competition of many objects concerned with single symbol recognition. A room filled with such objects manages input and sends one input character after another to the objects. 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 of these objects that entered 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 state; therefore, these objects can be more powerful recognizers than finite state automata. This permits recognition of things like nested comments. While the individual objects are small, problem-specific automata, the room combines them into a larger, non-deterministic system.

flex-like tools 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, however.

Building and reducing the automaton requires compilation. *oolex* avoids this because the symbol recognition objects save the states locally and the competition essentially operates them in parallel. If there is an ambiguity, e.g., if recognizers are optionally iterated, the system can generate additional instances of a recognizer and add them to the competition.

Partitioning a text does not make much sense unless there are actions to deal with the pieces. A symbol 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

oolex is based on version 1.1 of Sun's Java Development Kit because version 1.2 was not available on all platforms. Once a newer JDK is available universally `java.util.Vector` can be replaced by the more efficient `java.util.ArrayList`.

Input

`Input` manages character input to *oolex*:

```
public class Input {
     public Input (Reader in, int front, int rear) { ... }
     public Input (Reader in [], int front, int rear) { ... }
     ...
     protected char buffer [];
     ...
     public static class EmptyTokenException extends Exception { ... }
     public static class IllegalCharacterException extends Exception { ... }
     ...
     public boolean token (Scan entry) throws IOException, EmptyTokenException,
                                              IllegalCharacterException { ... }
     public String toString() { ... }
     public int line() { ... }
}
```

Given a stream `in`, an `Input` object reads many characters at once and knows the position of the next character in the buffer. If the position passes a low-water mark `front`, the buffer content is shifted. If the position reaches the the end of the buffer, at least `rear` characters are read into the buffer.

Calling `token()` starts a new round in the competition to recognize the symbol modeled by `entry`. This seems to connect `Input` only with a single symbol recognizer, but suitable implementations and subclasses of Scan will recruit more than one competitor.

The job of `token()` is to pass one input character after another to `entry` which will sooner or later indicate that it did or did not recognize a symbol. `token()` then either returns `true` or it throws an `IllegalCharacterException`. At end of file `token()` returns `false`.

An `Input` object can be constructed for a sequence of `Reader` objects. In this case, `token()` returns `false` once the end of the last `Reader` is reached. However, a symbol is not permitted to extend across more than one `Reader`.

`line()` produces the current line number within the current `Reader`. Only `Input` can produce this information because `Input` reads many characters into a buffer and effectively hides the position inside the buffer.

The `IllegalCharacterException` message contains the illegal character, the current line number and the description returned by `toString()` from the current `Reader`.

`Input`'s own `toString()` returns a description with the class name, the current line number and the description of the current `Reader`. The description can be manipulated if the user extends the `Reader` instances to provide suitable implementations of `toString()`.

`Input` solely employs `Reader` and is therefore based on Unicode. `Input` itself does not imply a character encoding because there is no constructor using an `InputStream`.

To summarize: `Input` manages character input to the room of objects implementing a symbol scanner.

Scan and the *Action* interface

Every symbol recognizer class must descend from `Scan`. This is an abstract class which requires subclasses to implement certain methods and implements some others:

```
public abstract class Scan implements Serializable, Cloneable {
     protected transient int length;
     public int getLength() { return length; }

     protected transient boolean accepted;
     public boolean getAccepted() { return accepted; }

     public abstract boolean next (int ch);

     public Object clone ();

     protected void action (char buf[], int off, int len) { ... }
     public Scan setAction (Action action) { ... }
     public Action getAction () { ... }

     public Object genValue(char buf [], int off, int len) { ... }

     public static Scan newRe (String pattern) { ... }  nicht erklaert!
}
```

`Input`'s method `token()` calls `next()` to send one character after another to a `Scan` object. `next()` returns `true` as long as the object wants to receive more characters and `false` otherwise.

The algorithm implemented by `next()` really has three results: it knows if the current character completed a symbol; it knows how many of the characters seen thus far, if any, are at most acceptable as a symbol; and it knows if it makes sense to accept more characters to discover a longer symbol. `next()` could have returned an aggregate or a bit-encoded int or long to describe all three results. Instead, `Scan` provides `accepted` and `length` to describe the first two results and lets `next()` itself return the third. For efficiency there is package and subclass access to `accepted` and `length` and there are access methods for other packages.

`token()` first sends `clone()` to each object and `clone()` returns an initialized object which performs the actual recognition.

Scan objects usually contain local state information. Therefore, clone() should generate a deep copy and initialize it for recognition. In particular, accepted and length must be initialized in clone(): if a Scan object recognizes an empty input, clone() sets accepted to true and length to zero, otherwise accepted is false and length is -1.

The implemenation of clone() in Scan defers to clone() in it's superclass Object to create a shallow copy of the receiver of the method, i.e., of the appropriate subclass of Scan, and then initializes accepted with false and length with -1 before it returns the copy. If a subclass of Scan overwrites clone(), it must defer to the superclass and initialize (as a deep copy) only it's own fields.

Action describes an action to be performed when a symbol is recognized. Action is a simple interface:

```
public interface Action extends Serializable {
     public void action(Scan sender, char buf[], int off, int len);
}
```

A Scan object has an Action object as a property that can be set with setAction() and queried with getAction(). The method setAction() returns this so that calls can be cascaded in initialization lists for arrays.

Once Input's method token() determines that it's Scan object entry has successfully recognized a symbol, it calls entry's method action() with the text of the symbol. Scan provides a dummy implementation of action(): if an Action object was set up with setAction(), the call is passed on, otherwise nothing happens.

Scan objects are looking for specific symbols. It seems likely that the objects know how to create an object to suitably represent the recognized text, e.g., a number recognizer constructs a Number object etc. Therefore, genValue() returns such an object, by default a String.

Alt

Alt manages a competition of symbol recognizers to discover the longest possible symbol thus completing the idea of a room full of objects for lexical analysis. A tie is decided in favour of the first recognizer:

```
public static class Alt extends Scan {
     public Alt (Scan[] entry) { ... }
     protected final Scan[] entry;    // match one (or first) of these
     protected transient int winner;  // index of first of max length

     public Object clone () { ... }
     public boolean next(int ch) { ... }
     public void action (char buf[], int off, int len) { ... }
}
```

Alt descends from Scan and can thus be used as argument for token(). Once a round is completed, winner indicates the successful recognizer in entry. Therefore, Alt completes the implementation of a room of objects to recognize symbols.

The argument to token() does not necessarily have to be an Alt object. Any Scan object can be specified so that it may be tested individually.

Alt overwrites action(): if an Alt object has it's own Action object, it sends action() there; otherwise it will send action() to entry[winner].

Alt overwrites clone(): it calls clone() in it's superclass Scan to create the new recognizer and calls clone() for each object in entry to produce a deep copy, it sets accepted and length according to the entries and returns the cloned recognizer.

Examples

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

```
public static class Set extends Scan {
     public Set (String set, boolean inside) {
          if (set == null) throw new IllegalArgumentException("null set");
          this.set = set; this.inside = inside;
     }
     protected final String set;
     protected final boolean inside;

     public boolean next (int ch) {
          length = (set.indexOf((char)ch) >= 0) == inside ? 1 : -1;
          accepted = length == 1; return false;
     }
}
```

Before scanning starts `accepted` is `false` and `length` is `-1` as set by `clone()` in `Scan`. A `Set` can accept only a single character; therefore, `next()` always returns `false`.

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

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

```
public static class SetMN extends Scan {
     public SetMN (String set, boolean inside, int m, int n) { ... }
     public Object clone () { ... }
     public boolean next (int ch) { ... }
}
```

The value of `m` can be zero, therefore, `SetMN` can recognize an empty input. As mentioned above, in this case `clone()` calls `super.clone()` and then sets `accepted` to `true` and `length` to 0.

`Input`'s method `token()` cannot be happy if a recognizer such as `SetMN` wins by recognizing no input, because further calls to `token()` would no longer consume input characters. Therefore, `token()` reacts with an `EmptyTokenException` containing the current line number and the description of the current `Reader`.

The user can catch the exception and change the recognition strategy. This is an advantage over *flex* where the user cannot interfere with the infinite loop.

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() { super("0123456789", true, 1, Integer.MAX_VALUE); }

     public Object genValue(char buf [], int off, int len) {
          return new Long(new String(buf, off, len));
     }
}
```

`Int` inherits `clone()` from `SetMN` which calls `clone()` in `Scan` to create the new object. Therefore, if `clone()` is sent to an `Int`, the method responds with a new `Int` object because `clone()` in `Scan`

creates an object of the receiver's class, in this case an `Int`. Thus, if `genValue()` is sent to the result of `clone()`, it will return a `Long`.

`Word` objects recognize character sequences.

`EOL` is a subclass of `Word` to recognize the platform dependent end of line character. Unlike *flex* this can be tailored at runtime.

`BOL` recognizes the beginning of a line. Unfortunately, this requires close cooperation with `Input`, but both classes are in the same package and the interaction can remain hidden.

A very powerful class `Balanced` recognizes a character sequence delimited by a `Word` at the beginning and a different `Word` at the end. The sequences can even be nested. For simplicity, the constructor checks that the delimeters do not start with the same character.

`Balanced` is the base class for comment recognition, for example:

```
public class CComment extends Balanced {
     public CComment(boolean nested) { super(new Word("/*"), new Word("*/"), nested); }
}
```

Ignore symbols

A scanner for a compiler often ignores some recognized symbols; for example whitespace or comment. To support this in a simple way `Scan` has two further methods:

```
public abstract class Scan implements Serializable, Cloneable {
     ...
     public final Scan setIgnore(boolean ignore) { ... }
     protected boolean getIgnore() { ... }
     ...
}
```

After inside `token()` a symbol was found and `action()` was send, the winner is asked through `getIgnore()` if this symbol should be ignored. If yes, `token()` searches for the next symbol.

`setIgnore()` just sets the return value of `getIgnore()` and returns `this` so that calls can be cascaded in initialization lists for arrays.

If `token()` is used with a `Alt`, a call to `getIgnore()` must be forwarded to the winning alternative:

```
public static class Alt extends Scan {
     ...
     protected boolean getIgnore() {
          return entry[winner].getIgnore();
     }
     ...
}
```

Regular Expressions

oolex was designed so that specialized classes should take care of symbol recognition; however, it only takes two or three more container classes similar to `Alt` to implement a `Scan` subclass for regular expressions.

`Seq` contains two `Scan` objects, `first` and `second`, which must be matched one after another. Longer sequences can be modeled by cascading `Seq` objects. Implementing `next()` is complicated by the fact that depending on `first` several copies of `second` may have to be activated in parallel, but `Seq` can generate them on the fly using `clone()`.

Loop objects recognize between m and n iterations of a subsidiary Scan object.

Opt is a simplified version of Loop that lets it's Scan object optionally perform recognition, i.e., m is zero and n is one in this case.

Scan implements a class method newRe() which accepts en egrep-like pattern as a string and returns a tree representing the pattern as a Scan object. The tree uses Alt, Loop, Opt, and Seq as containers and BOL, EOL, Set, SetMN, and Word as leaves.

All the classes for the regular expressions are public; therefore, different parsers may be used to map different representations of regular expressions to Scan trees. The classes are an API for regular expressions based on object trees and as such could be the subject of another paper.

Base Classes

Scan contains the essential recognizers as statically nested classes:

```
public abstract class Scan implements Serializable {
     ...
     public interface Action extends Serializable { ... }
     public static class Alt extends Scan { ... }
     public static class Seq extends Scan { ... }
     public static class Opt extends Scan { ... }
     public static class Loop extends Scan { ... }
     public static class Set extends Scan { ... }
     public static class SetMN extends Scan { ... }
     public static class Word extends Scan { ... }
     public static class EOL extends Word { ... }
     public static class BOL extends Scan { ... }
}
```

Library

Scanners are primarily used in compiler construction. A number of classes, some mentioned above, have already been implemented to support typical languages. The names are rather descriptive:

```
Balanced          CComment          CIdentifier       Char
Flt               HashComment       HexNumber         Int
JavaIdentifier    JavaWhitespace    JavadocComment    OctalNumber
QuotedString      SlashSlashComment
```

Serialization

Scan and Action include the interface Serializable. The classes implemented thus far ensure that only necessary instance variables, such as the permissible characters in Set or the character sequence in Word, are serialized.

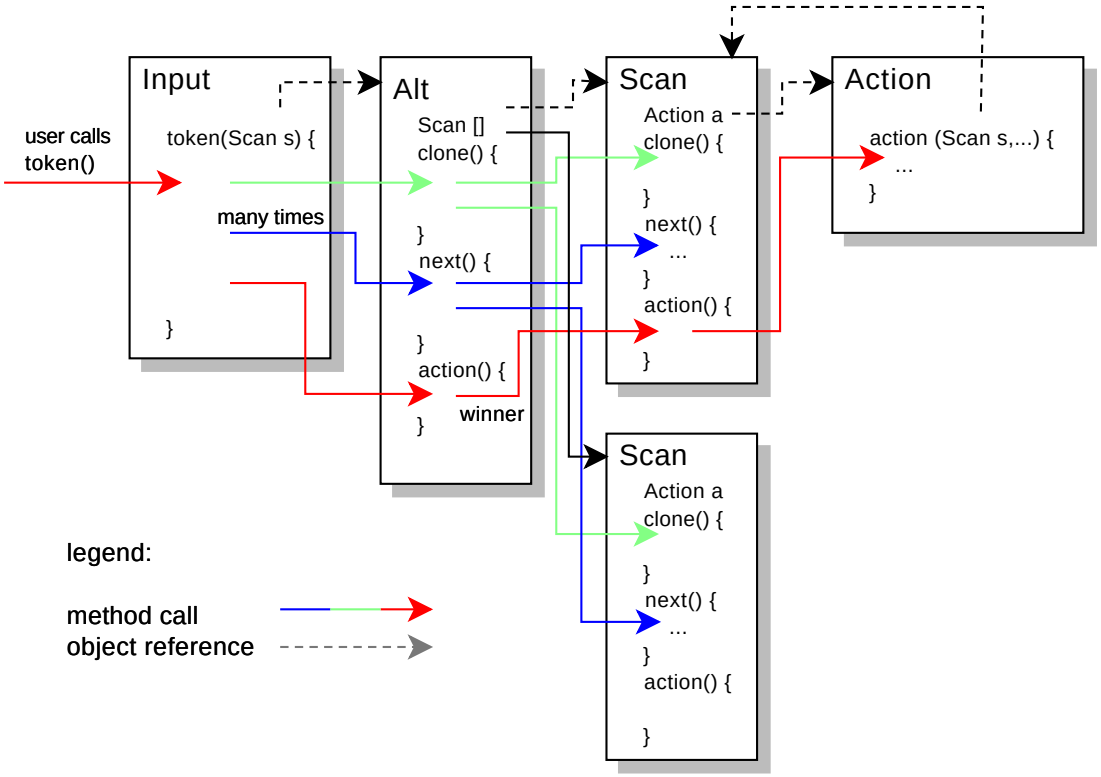
A scanner can be stored as a collection of serialized objects and no explicit construction is required whenever the scanner is read in to be executed. Conceivably, different scanners might even share the same serialized objects.

Only objects returned by clone() perform recognition, i.e., clone() can be used to activate an object after serialization.

Control flow

The following diagram traces the control flow during symbol recognition. The dashed connections indicate how the objects are connected: When token() is called Input receives an Alt which

controls two Scan objects; the first of these references an Action object. As an abstract class Scan cannot be instantiated; however, the exact subclass is not relevant for the control flow discussion.



`token()` first sends `clone()` to it's argument. `Alt` forwards `clone()` to each of it's constituents.

Following initialization `token()` uses `next()` to send one input character after another to `Alt` which forwards them to it's still active scanners. Once the last scanner quits, `accepted` and `length` are set and `Alt` returns `false` as the result of `next()`.

`token()` now checks `accepted` at `Alt` to find out if something was recognized. If not, `token()` throws an `IllegalCharacterException`. On the other hand, if `length` at `Alt` is zero, `token()` throws an `EmptyTokenException`.

Otherwise `token()` sends `action()`, and because `Alt` does not have it's own `Action` instance, it forwards the message to the winning recognizer. The drawing assumes that this Scan object references an `Action` object which finally receives the `action()` message.

At last `token()` terminates with `true`. If end of file was reached and `token()` is called again, it returns `false`.

oolex scanners can be text filters or parts of compilers. In the first case `action()` must be implemented to perform text conversion, in the latter case `action()` could send `genValue()` to it's caller to receive an object corresponding to the recognized character sequence. In a compiler this object would convey semantic information to parser and code generation.

Analysis

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

If the class library 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 table. However, given a reasonable library it should hardly be necessary. Moreover, oolex does provide a way to generate a `Scan` tree to implement a regular expression.

The user has to understand how `Input` is used and what subclasses of `Scan` have been implemented. Learning how to use a tool like *flex* and the subtleties of regular expression is likely to take longer.

An *oolex* scanner needs the class library at runtime, a *flex* scanner technically is a standalone piece of software. However, the class libraries simply are 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 *flex* scanner is a deterministic finit state automaton described by a state table. An *oolex* scanner is not deterministic and runs an `Alt` object to follow all possibilities in parallel. A `Seq` object might operate several of it's constituents in parallel. Therefore, an *oolex* scanner is almost certainly slower.

JLex implements a *flex* scanner in Java. Looking for typical symbols in a C source file with 370 kb takes four times longer using *oolex* than using *JLex*. Most *oolex* scanners can be frozen and converted into *JLex* scanners: where feasible we have added methods to our `Scan` classes to emit *JLex* patterns and actions to call on our serialized `Action` objects. Freezing eliminates flexibility but it increases performance.

oolex' object oriented approach has a lot of advantages and some additional features. It is certainly more intuitive for object-oriented languages such as Java, Objective C or C++.

`Input` only manages characters. Each call of `token()` can provide a different `Scan` tree and new `Scan` objects could even be added to `Alt` or `Seq` right during execution. This corresponds to defining states and adding new regular expressions to a *flex* table; however, it does require a complete development cycle — edit, preprocess, compile — before the revised *flex* scanner can be used.

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

The class library contains simple but powerful recognizer classes. For *flex* 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 expression have to be copied manually.

A vendor can provide it's own class library for *oolex*. 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 `Alt` could decide on a different approach to resolving ambiguity. This is not possible for *flex*.

`Input` uses Unicode characters. If *JLex* is forced to use Unicode it takes much longer to create the scanner.

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

Objects encapsulate state; therefore, `Scan` objects can be more powerful recognizers than plain finite state automata, viz the recognition of nested comments. Most objects are very simple, however; with `Alt` they are combined to form a much larger, non-deterministic system.

Conclusion

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

We think that oolex provides many advantages over purely regular expression based scanners.There is a performance penalty but it is justified by flexibility, a shorter development cycle and more features.

oolex can compile a regular expression into a tree of recognizer objects, so no expressive power is lost. However, oolex is much easier to use because there is already a library of recognizers for most typical symbol patterns. Some of these recognizers are more powerful than regular expressions.

To summarize, oolex 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 JLex: `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

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

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

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

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

[8] *Oolex* sources and/or homepage *oolex*: See `http://www.inf.uos.de/toDo` or send a mail to `bernd.kuehl@informatik.uni-osnabrueck.de`

toDO

Homepage oolex und Reference 8 anpassen.