Project Jauk

Jauk exports a text parsing framework from the Gap Data + LXL system. This framework employs regular expressions and object oriented programming for object oriented parsing. The following example illustrates one such application (from the Gap Data ODL).

```
package gap.odl;
import jauk.Re;
import jauk.Scanner;

public class Class {
    private final static Re Open = new Re("<_>class ~(\\{)\\{<_>");

    public final String name;

    public Class(Scanner reader){
        super();
        String input = reader.next(Open);
        if (null != input){
            StringTokenizer strtok = new StringTokenizer(input," \t\r\n{");
            strtok.nextToken();
            this.name = strtok.nextToken();
        }
        else
            throw new SyntaxException("Class definition not found");
    }
}
```

Jauk employed Java Regular Expressions (and Scanner) initially. However, the Perl style nondeterministic automata were found to be desperately lacking of basic (Grep or Awk) utility. As a result, "Project Jauk" went searching for -- and found -- a suitable replacement.

Project Automaton by Anders Møller presents a facility oriented to a suite of applications employed at BRICS. For the purposes of Project Jauk, some modifications have been made.

This work was motivated by a few interesting regular expressions that failed to perform as desired. Automaton includes high utility features such as Named Automata, Complements, and Strings. From the grammatical definition of Automaton, one sees a powerful equivalence between Simple Expressions and Character Classes, and the applicability of both to operation under the Complement operator.

The continuation problem

These facilities permit an elegant solution to the classic problem of the multiline comment from the C Programming Language,

```
/*
 * C multiline comment
 */
```

matching to a regular expression as in the following example.

```
/\*~(\*/)*\*/
```

(Or equivalently, using Automaton's strings

```
"/*"~("*/")*"*/"
```

).

While this expression works to match the string, above, it will continue to match any number of such strings (multiline comments) found in the source text (i.e. a practical software source file having many of these comments).

This was a problem for Jauk, and forced the issue of forking Automaton into a version for Jauk.

For the convenience of effort, Automaton was condensed into a minimal, literate programming style. Most comments where stripped. Qualified names where truncated. Classes were reorganized. And generally the package (as book) made more welcoming to the reader. From this point, the final stages of the adaptation of Automaton to Jauk began.

With the comprehension of Automaton within reach, the continuation problem (described above) was developed.

First to refine our comprehension of regular expressions, the complement + anti-complement pattern has no reason to terminate when compiled into an automaton. As every possible input character is a state transition of the automaton, there is no exit transition.

I use regular expressions many times per day in exacting applications, however this degree of knowledge and experience is only a vague recognition of the operation of the automata produced by a regular expression library. For example, in Grep or Sed or Awk the kind of boolean matching implemented by the original Automaton is employed. Realizing the distinction to the offset parsing required by Jauk was among my first steps in solving the continuation problem.

Another was the realization that the character state map nature of the automaton is exceedingly simple, and thereby powerful, and thereby impossible to modify according to the usual logic of systems programming. Very often one makes formerly lost information distinct to improve the design of a software system. As a software engineer, my brain is extremely effective in performing this operation. And this skill was useless, here, for a fairly subtle reason.

It would seem that the end of the regular expression would be a sensible place for the automaton to halt. In a more complex software system than a minimal deterministic finite state

machine this would be an obvious case of having lost the information implied by the end of the regular expression. However, the regular expression that is a complement and anti complement describes a finite state machine with no exit. There is no character that is not included in a transition of the automaton.

So, Jauk needed to add some facility for a halting state. Happily, Automaton had lit the path by being a set theoretically complete language. Praise the benefits of completeness.

The empty set (rather “language” in the “syntax” context) had been employed in Automaton as a reject state -- for purposes remaining to be explored.

The reject state is no more than the intermediate state of a running automaton such as `"a.*z"`. The accept state is likewise common while running the automaton produced for a regular expression such as `"[a-zA-Z0-9]*"`. Therefore these states are useful in their changes from reject to accept, and from accept to reject.

For Jauk the empty set defined from a regular expression has been changed to an accept state.

Automaton describes the empty set by the pound (#) character.

The positive empty set solves the continuation problem in the C comment pattern as follows.

```
"/*"~("*/")*"*/"#
```

This expression transforms the last state of the automaton into an accepting state, permitting the use of the automaton to make distinct a transition from accepted to rejected states as the halting case.

Named Automata

Along the way the named automata facility from the original Automaton package has been refactored into a facility that is easy to replace or extend. This is in the class named *NamedAutomata* in the *automaton* package of Jauk.

Serializable Automata

The former facility for serializable automata has been dropped in favor of a pair of ideas. The first is the string representation of a regular expression is the primary form. And the second is that a serialized automaton would be an instance of the *Compiled* class, as a subclass of the *Automaton* class (in future). This second idea would maximize the efficiency of the serialized data (perhaps at the expense of reversibility to the string representation of the regular expression), however the utility of serialized automata is seen as small enough that the work has not been performed as of this date.

Additional modifications

A few additional changes have been made to Automaton for Jauk in support of the offset parsing rather than boolean matching purposes of Jauk. These are confined to the class named *Compiled* in the *automaton* package of Jauk.

Author

John Pritchard
Syntelos